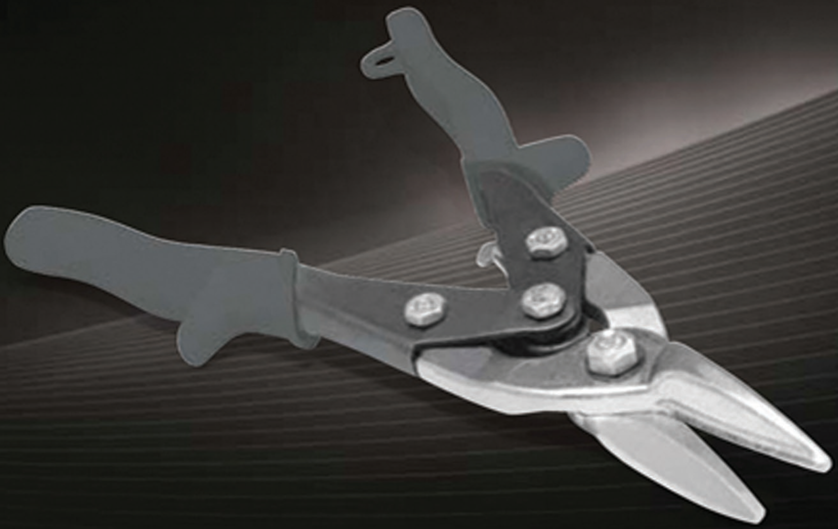
Microsoft

Foreword by Lubor Kollar
Group Program Manager, SQL Server Customer Advisory Team

Inside Microsoft® SQL Server® 2008: T-SQL Programming



Itzik Ben-Gan
Dejan Sarka, Roger Wolter,
Greg Low, Ed Katibah, Isaac Kunen
Kalen Delaney–Series Editor

SOLID QUALITY MENTORS

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2009932820

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9   QWT   4 3 2 1 0 9

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Acquisitions Editor: Ken Jones
Developmental Editor: Denise Bankaitis
Project Editor: Denise Bankaitis
Editorial Production: Ashley Schneider, S4Carlisle Publishing Services
Technical Reviewer: Steve Kass; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X15-74121

To my siblings, Ina & Mickey

—Itzik

Table of Contents

Foreword

Let me start with a few words about the lead author of this book, Itzik Ben-Gan. He is a mentor, consultant, presenter, teacher, and writer. All his roles have a common theme—Microsoft SQL Server programming. But wait—there's even more: he is an MVP (officially "Microsoft Valued Professional," but often interpreted as Most Valuable Programmer) and a close partner with many SQL Server developers in Redmond, Washington. The combination of these traits puts Itzik into a unique position when writing a book about SQL Server programming. He knows what works and what does not. He knows what performs and what does not. He knows what questions people ask when he is teaching. And he knows what people grasp easily and what they don't when it comes to introducing more complex concepts of SQL Server programming.

Itzik invited several coauthors to write parts of this book. He does not hesitate to tap the resources of the SQL Server development team when it comes to introducing the newest SQL Server technologies. This was the case with spatial data chapter written by Ed Katibah (a.k.a. "Spatial Ed") and Isaac Kunen and with the Service Broker chapter written by Roger Wolter. Dejan Sarka helped with CLR and XML, and contributed the fascinating chapter on temporal support in the relational model, where he pokes at SQL Server developers about usefulness of PACK and UNPACK relational operators still missing in SQL Server. Greg Low untangled the many ways one can go about tracking access and changes to data and metadata. Both Dejan and Greg are SQL Server veterans and Itzik's colleagues in Solid Quality Mentors.

I personally believe in hands-on learning when it comes to programming. This book has many examples and they are all presented in a way that allows you to run them on your own SQL Server installation. If you don't have your own SQL Server installation, you can go to *http://www.microsoft.com/sql* and download the evaluation version of SQL Server 2008 (you must have a Windows Live ID; the evaluation version is Enterprise and it is good for 180 days). Preferably you should be using the Enterprise or Developer edition of SQL Server to run the examples. And no, you don't need to retype all code segments in the book! You can download the source code from *http://www.InsideTSQL.com*.

If you are new to the SQL language you should start with the earlier published book, *Microsoft SQL Server 2008: T-SQL Fundamentals*. If you are new to SQL Server but you have used other SQL supporting products you may want to start with the companion book *Inside Microsoft SQL Server 2008: T-SQL Querying*. But you can jump right into this book as well; it will give you great insight into SQL Server–specific programming. You can use the examples in the book to find out whether you need to study specific statements where SQL Server has a different implementation from your previous experiences and you can use these books for reference.

Even if you are a seasoned SQL Server developer I'm sure this book will show you new and more efficient ways to perform your tasks. For example, I agree with Dejan that there are few CLR UDTs in production systems. And this is not only true for UDTs—few UDFs, triggers, and stored procedures are written in CLR languages. The book provides numerous examples of C# and Microsoft Visual Basic solutions. Most of the examples are presented in both C# and Visual Basic, which are the most popular CLR languages. The authors are careful about CLR recommendations because of performance implications. Itzik not only provides general performance guidelines, but he also tells you how long the alternatives took to execute on his computer. Of course, you will try it on your computer!

Performance considerations are not restricted to CLR. You will find performance improvement tips in every single chapter of this book. For example, in Chapter 7, "Temporary Tables and Table Variables," you will learn when it is better to use temporary tables and when it is better to use table variables. Itzik uses simple examples, interpreting query plans and showing how to use IO counters when comparing different solutions for the same task.

I mentioned that Chapter 12— Dejan's "Temporal Support in the Relational Model" chapter—is fascinating. Why? Let me share a little secret. Some time ago we considered implementing special support for temporal data inside SQL Server. The work was intense and the SQL Server development team got help from leading academic sources as well. One development lead even personalized the license plate on his car to "TIME DB." What happened with the project? The implementation was complex and costly. Some of the alternatives were repeatedly re-evaluated without providing a clear winner. And there was always a counter-argument—"you can use a workaround." Whenever this argument was challenged someone wrote a piece of code showing how a particular temporal task could be achieved using *existing* features in SQL Server. But I don't know anybody who did as complete a job as Dejan in Chapter 12 of this book!

I worked with Roger Wolter on the same team when he was responsible for developing the brand new Service Broker in SQL Server 2005. His chapter (Chapter 16) is great reflection of his personality—deep with very accurate details in perfect structure. If you are new to Service Broker you may want to start reading this chapter from the end, where you will learn which scenarios you can use Service Broker with, along with a brief comparison of Service Broker with messaging solutions delivered by Microsoft Message Queue (MSMQ), BizTalk, and Windows Communication Foundation (WCF). Bank Itau in Brazil and MySpace are two examples of SQL Server customers who use Service Broker for very different purposes. Bank Itau uses Service Broker for batch processing. In MySpace, Service Broker creates a communication fabric among hundreds of SQL Servers behind the MySpace.com social networking site.

I'm confident you will find this book useful and worth reading whether you are a new or seasoned SQL Server user. It is an invaluable reference for developers, data architects, and administrators.

Lubor Kollar
Group Program Manager
SQL Server Customer Advisory Team
Microsoft, Redmond, Washington U.S.A.

Acknowledgments

Several people contributed to the T-SQL Querying and T-SQL Programming books and I'd like to acknowledge their contributions. Some were involved directly in writing or editing the books, whereas others were involved indirectly by providing advice, support, and inspiration.

To the coauthors of *Inside Microsoft SQL Server 2008: T-SQL Querying*—Lubor Kollar, Dejan Sarka, and Steve Kass; and to the coauthors of *Inside Microsoft SQL Server 2008: T-SQL Programming*—Dejan Sarka, Roger Wolter, Greg Low, Ed Katibah, and Isaac Kunen, it is a great honor to work with you. It is simply amazing to see the level of mastery that you have over your areas of expertise, and it is pure joy to read your texts. Thanks for agreeing to be part of this project.

To Lubor, besides directly contributing to the books by writing a chapter for T-SQL Querying and the foreword to *T-SQL Programming,* you provide support, advice, friendship, and are a great source of inspiration. I always look forward to spending time with you—hiking, drinking, and talking about SQL and other things.

To Dejko, your knowledge of the relational model is admirable. Whenever we spend time together I learn new things and discover new depths. I like the fact that you don't take things for granted and don't blindly follow the words of those who are considered experts in the field. You have a healthy mind of your own, and see things that very few are capable of seeing. I'd like to thank you for agreeing to contribute texts to the books. I'd also like to thank you for your friendship; I always enjoy spending time with you. We need to do the beer list thing again some time. It's been almost 10 years!

To the technical editor of the books, Steve Kass, your unique mix of strengths in mathematics, SQL, and English are truly extraordinary. I know that editing both books and also writing your own chapters took their toll. Therefore I'd like you to know how much I appreciate your work. I know you won't like my saying this, but it is quite interesting to see a genius at work. It kept reminding me of Domingo Montoya's work on the sword he prepared for the six-fingered man from William Goldman's *The Princess Bride*.

To Umachandar Jayachandran (UC), many thanks for helping out by editing some of the chapters. Your mastery of T-SQL is remarkable, and I'm so glad you could join the project in any capacity. I'd also like to thank Bob Beauchemin for reviewing the chapter on spatial data. I enjoy reading your texts; your insights on SQL Server programmability are always interesting and timely.

To Cesar Galindo-Legaria, I feel honored that you agreed to write the foreword for the *T-SQL Querying* book. The way you and your team designed SQL Server's optimizer is simply a marvel. I'm constantly trying to figure out and interpret what the optimizer does, and whenever I manage to understand a piece of the puzzle, I find it astonishing what a piece of

software is capable of. Your depth of knowledge, your pleasant ways, and your humility are an inspiration.

To the team at Microsoft Press: Ken Jones, the product planner: I appreciate the personal manner in which you handle things, and always look forward to Guinness sessions with you. I think that you have an impossible job trying to make everyone happy and keep projects moving, but somehow you still manage to do it.

To Sally Stickney, the development editor, thanks for kicking the project off the ground. I know that the T-SQL Querying book was your last project at Microsoft Press before you started your new chosen path in life, and am hopeful that it left a good impression on you. I wish you luck and happiness in your new calling.

To Denise Bankaitis, the project editor, you of all people at Microsoft Press probably spent the most time working on the books. Thanks for your elegant project management, and for making sure things kept flowing. It was a pleasure to work with you.

I'd also like to thank DeAnn Montoya and Ashley Schneider, the project managers for the vendor editorial team, S4Carlisle, and Becka McKay, the copy editor. I know you spent countless hours going over our texts, and I appreciate it a lot.

To Solid Quality Mentors, being part of this amazing company and group of people is by far the best thing that happened to me in my career. It's as if all I did in my professional life led me to this place where I can fulfill my calling, which is teaching people about SQL. To Fernando Guerrero, Brian Moran, Douglas McDowell: the company grew and matured because of your efforts, and you have a lot to be proud of. Being part of this company, I feel a part of something meaningful, and that I'm among family and friends—among people that I both respect and trust.

I'd like to thank my friends and colleagues from the company: Ron Talmage, Andrew J. Kelly, Eladio Rincón, Dejan Sarka, Herbert Albert, Fritz Lechnitz, Gianluca Hotz, Erik Veerman, Jay Hackney, Daniel A. Seara, Davide Mauri, Andrea Benedetti, Miguel Egea, Adolfo Wiernik, Javier Loria, Rushabh Mehta, Greg Low, Peter Myers, Randy Dyess, and many others. I'd like to thank Jeanne Reeves and Glen McCoin for making many of my classes possible, and all the back-office team for their support. I'd also like to thank Kathy Blomstrom for managing our writing projects and for your excellent edits.

I'd like to thank the members of the SQL Server development team that are working on T-SQL and its optimization: Michael Wang, Michael Rys, Eric Hanson, Umachandar Jayachandran (UC), Tobias Thernström, Jim Hogg, Isaac Kunen, Krzysztof Kozielczyk, Cesar Galindo-Legaria, Craig Freedman, Conor Cunningham, Yavor Angelov, Susan Price, and many others. For better or worse, what you develop is what we have to work with, and so far the results are outstanding! Still, until we get a full implementation of the OVER clause, you know I won't stop bothering you.

I'd like to thank Dubi Lebel and Assaf Fraenkel from Microsoft Israel, and also Ami Levin who helps me run the Israeli SQL Server users group.

To the team at *SQL Server Magazine*: Megan Bearly, Sheila Molnar, Mary Waterloo, Michele Crockett, Mike Otey, Lavon Peters, Anne Grubb; being part of this magazine is a great privilege. Congratulations on the tenth anniversary of the magazine! I can't believe that 10 years passed so quickly, but that's what happens when you have fun.

To my fellow SQL Server MVPs: Erland Sommarskog, Alejandro Mesa, Aaron Bertrand, Tibor Karaszi, Steve Kass, Dejan Sarka, Roy Harvey, Tony Rogerson, Marcello Poletti (Marc), Paul Randall, Bob Beauchemin, Adam Machanic, Simon Sabin, Tom Moreau, Hugo Kornelis, David Portas, David Guzman, Paul Nielsen, and many others: Your contribution to the SQL Server community is remarkable. Much of what I know today is thanks to our discussions and exchange of ideas.

To my fellow SQL Server MCTs: Tibor Karaszi, Chris Randall, Ted Malone, and others: We go a long way back, and I'm glad to see that you're all still around in the SQL teaching community. We all share the same passion for teaching. Of anyone, you best understand the kind of fulfillment that teaching can bestow.

To my students: Without you my work would be meaningless. Teaching is what I like to do best, and the purpose of pretty much everything else that I do with SQL—including writing these books—is to support my teaching. Your questions make me do a lot of research, and therefore I owe much of my knowledge to you.

To my parents, Emilia and Gabriel Ben-Gan, and to my siblings, Ina Aviram and Michael Ben-Gan, thanks for your continuous support. The fact that most of us ended up being teachers is probably not by chance, but for me to fulfill my calling I end up traveling a lot. I miss you all when I'm away and I always look forward to our family reunions when I'm back.

To Lilach, you're the one who needs to put up with me all the time, and listen to my SQL ideas that you probably couldn't care less about. It's brainwashing, you see—at some point you will start asking for more, and before you know it you will even start reading my books. Not because I will force you, rather because you will want to, of course. That's the plan, at least… Thanks for giving meaning to what I do, and for supporting me through some rough times of writing.

Introduction

This book and its prequel—*Inside Microsoft SQL Server 2008: T-SQL Querying*—cover advanced T-SQL querying, query tuning, and programming in Microsoft SQL Server 2008. They are designed for experienced programmers and DBAs who need to write and optimize code in SQL Server 2008. For brevity, I'll refer to the books as *T-SQL Querying* and *T-SQL Programming*, or just as *these books*.

Those who read the SQL Server 2005 edition of the books will find plenty of new material covering new subjects, new features, and enhancements in SQL Server 2008, plus revisions and new insights about the existing subjects.

These books focus on practical common problems, discussing several approaches to tackle each. You will be introduced to many polished techniques that will enhance your toolbox and coding vocabulary, allowing you to provide efficient solutions in a natural manner.

These books unveil the power of set-based querying, and they explain why it's usually superior to procedural programming with cursors and the like. At the same time, they teach you how to identify the few scenarios where cursor-based solutions are superior to set-based ones.

The prequel to this book—*T-SQL Querying*—focuses on set-based querying and query tuning, and I recommend that you read it first. This book—*T-SQL Programming*—focuses on procedural programming and assumes that you read the first book or have sufficient querying background.

T-SQL Querying starts with five chapters that lay the foundation of logical and physical query processing required to gain the most from the rest of the chapters in both books.

The first chapter covers logical query processing. It describes in detail the logical phases involved in processing queries, the unique aspects of SQL querying, and the special mind-set you need to adopt to program in a relational, set-oriented environment.

The second chapter covers set theory and predicate logic—the strong mathematical foundations upon which the relational model is built. Understanding these foundations will give you better insights into the model and the language. This chapter was written by Steve Kass, who was also the main technical editor of these books. Steve has a unique combination of strengths in mathematics, computer science, SQL, and English that make him the ideal author for this subject.

The third chapter covers the relational model. Understanding the relational model is essential for good database design and helps in writing good code. The chapter defines relations and tuples and operators of relational algebra. Then it shows the relational model from a different perspective called *relational calculus*. This is more of a business-oriented perspective, as the logical model is described in terms of predicates and propositions. Data integrity is crucial for transactional systems; therefore, the chapter spends time discussing all kinds of constraints. Finally, the chapter introduces normalization—the formal process of improving database design. This chapter was written by Dejan Sarka. Dejan is one of the people with the deepest understanding of the relational model that I know.

The fourth chapter covers query tuning. It introduces a query tuning methodology we developed in our company (Solid Quality Mentors) and have been applying in production systems. The chapter also covers working with indexes and analyzing execution plans. This chapter provides the important background knowledge required for the rest of the chapters in both books, which as a practice discuss working with indexes and analyzing execution plans. These are important aspects of querying and query tuning.

The fifth chapter covers complexity and algorithms and was also written by Steve Kass. This chapter particularly focuses on some of the algorithms used often by the SQL Server engine. It gives attention to considering worst-case behavior as well as average case complexity. By understanding the complexity of algorithms used by the engine you can anticipate, for example, how the performance of certain queries will degrade when more data is added to the tables involved. Gaining a better understanding of how the engine processes your queries equips you with better tools to tune them.

The chapters that follow delve into advanced querying and query tuning, addressing both logical and physical aspects of your code. These chapters cover the following subjects: subqueries, table expressions, and ranking functions; joins and set operations; aggregating and pivoting data; TOP and APPLY; data modification; querying partitioned tables; and graphs, trees, hierarchies, and recursive queries.

The chapter covering querying partitioned tables was written by Lubor Kollar. Lubor led the development of partitioned tables and indexes when first introduced in the product, and many of the features that we have today are thanks to his efforts. These days Lubor works with customers that have, among other things, large implementations of partitioned tables and indexes as part of his role in the SQL Server Customer Advisory Team (SQL CAT).

Appendix A covers logic puzzles. Here you have a chance to practice logical puzzles to improve your logic skills. SQL querying essentially deals with logic. I find it important to practice pure logic to improve your query problem-solving capabilities. I also find these puzzles fun and challenging, and you can practice them with the entire family. These puzzles are a compilation of the logic puzzles that I covered in my T-SQL column in *SQL Server Magazine*. I'd like to thank *SQL Server Magazine* for allowing me to share these puzzles with the book's readers.

This book—*T-SQL Programming*—focuses on programmatic T-SQL constructs and expands its coverage to treatment of XML and XQuery, and the CLR integration. The book's chapters cover the following subjects: views, user-defined functions, stored procedures, triggers, transactions and concurrency, exception handling, temporary tables and table variables, cursors, dynamic SQL, working with date and time, CLR user-defined types, temporal support in the relational model, XML and XQuery (including coverage of open schema), spatial data, tracking access and changes to data, and Service Broker.

The chapters covering CLR user-defined types, temporal support in the relational model, and XML and XQuery were written by Dejan Sarka. As I mentioned, Dejan is extremely knowledgeable in the relational model, and has very interesting insights into the model itself and the way the constructs that he covers in his chapters fit in the model when used sensibly.

The chapter about spatial data was written by Ed Katibah and Isaac Kunen. Ed and Isaac are with the SQL Server development team, and led the efforts to implement spatial data support in SQL Server 2008. It is a great privilege to have this chapter written by the designers of the feature. Spatial data support is new to SQL Server 2008 and brings new data types, methods, and indices. This chapter is not intended as an exhaustive treatise on spatial data nor an encyclopedia of every spatial method which SQL Server now supports. Instead, this chapter will introduce core spatial concepts and provide the reader with key programming constructs necessary to successfully navigate this new feature to SQL Server.

The chapter about tracking access and changes to data was written by Greg Low. Greg is a SQL Server MVP and the managing director of SolidQ Australia. Greg has many years of experience working with SQL Server—teaching, speaking, and writing about it—and is highly regarded in the SQL Server community. This chapter covers extended events, auditing, change tracking, and change data capture. The technologies that are the focus of this chapter track access and changes to data and are new in SQL Server 2008. At first glance, these technologies can appear to be either overlapping or contradictory and the best use cases for each might be far from obvious. This chapter explores each technology, discusses the capabilities and limitations of each, and explains how each is intended to be used.

The last chapter, which covers Service Broker (SSB) was written by Roger Wolter. Roger is the program manager with the SQL Server development team and led the initial efforts to introduce SSB in SQL Server. Again, there's nothing like having the designer of a component explain it in his own words. The "sleeper" feature of SQL Server 2005 is now in production in a wide variety of applications. This chapter covers the architecture of SSB and how to use SSB to build a variety of reliable asynchronous database applications. The SQL 2008 edition adds coverage of the new features added to SSB for the SQL Server 2008 release and includes lessons learned and best practices from SSB applications deployed since the SQL Server 2005 release. The major new features are Queue Priorities, External Activation, and a new SSB troubleshooting application that incorporates lessons the SSB team learned from customers who have already deployed applications.

Hardware and Software Requirements

To practice all the material in these books and run all code samples it is recommended that you use Microsoft SQL Server 2008 Developer or Enterprise edition, and Microsoft Visual Studio 2008 Professional or Database edition. If you have a subscription to MSDN, you can download SQL Server 2008 and Visual Studio 2008 from *http://msdn.microsoft.com*. Otherwise, you can download a 180-day evaluation copy of SQL Server 2008 Enterprise edition free from: *http://www.microsoft.com/sqlserver/2008/en/us/trial-software.aspx*, and a 90-day free trial of Visual Studio 2008 Professional edition from: *http://www.microsoft.com/visualstudio/en-us/try/default.mspx*.

You can find the system requirements for SQL Server 2008 at the following link: *http://msdn.microsoft.com/en-us/library/ms143506.aspx*, and for Visual Studio 2008 at the following link: *http://www.microsoft.com/visualstudio/en-us/products/default.mspx*.

Companion Content and Sample Database

These books feature a companion Web site that makes available to you all the code used in the books, the errata, additional resources, and more. The companion Web site is *http://www.insidetsql.com*.

For each of these books the companion Web site provides a compressed file with the book's source code, a script file to create the books' sample database, and additional files that are required to run some of the code samples.

After downloading the source code, run the script file InsideTSQL2008.sql to create the sample database InsideTSQL2008, which is used in many of the books' code samples. The data model of the InsideTSQL2008 database is provided in Figure I-1 for your convenience.

Find Additional Content Online

As new or updated material becomes available that complements your books, it will be posted online on the Microsoft Press Online Developer Tools Web site. The type of material you might find includes updates to books content, articles, links to companion content, errata, sample chapters, and more. This Web site is available at *http://microsoftpresssrv.libredigital.com/serverclient/* and is updated periodically.

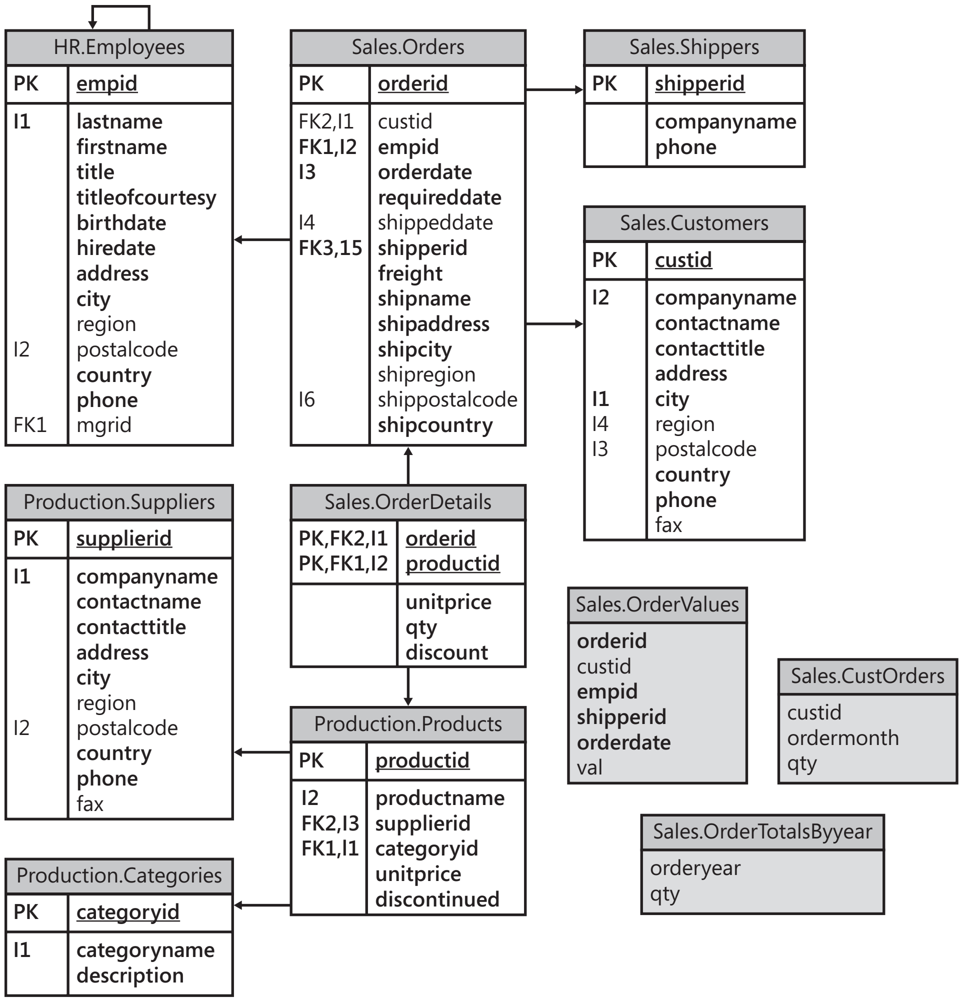


FIGURE I-1 Data model of the InsideTSQL2008 database

Support for These Books

Every effort has been made to ensure the accuracy of these books and the contents of the companion Web site. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books at the following Web site:

http://www.microsoft.com/learning/support/books/.

Questions and Comments

If you have comments, questions, or ideas regarding the books, or questions that are not answered by visiting the sites listed previously, please send them to me via e-mail to

itzik@SolidQ.com

or via postal mail to

Microsoft Press
Attn: *Inside Microsoft SQL Server 2008: T-SQL Querying and Inside Microsoft SQL Server 2008: T-SQL Programming* Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

Chapter 3
Stored Procedures

Itzik Ben-Gan and Dejan Sarka

Stored procedures are executable server-side routines. They give you great power and performance benefits if used wisely. Unlike user-defined functions (UDFs), stored procedures are allowed to have side effects; that is, they are allowed to change data in tables, and even alter object definitions. Stored procedures can be used as a security layer. You can control access to objects by granting execution permissions on stored procedures and not to underlying objects. You can perform input validation in stored procedures, and you can use stored procedures to allow activities only if they make sense as a whole unit, as opposed to allowing users to perform activities directly against objects.

Stored procedures also give you the benefits of encapsulation; if you need to change the implementation of a stored procedure because you developed a more efficient way to achieve a task, you can issue an ALTER PROCEDURE statement. As long as the procedure's interface remains the same, the users and the applications are not affected. On the other hand, if you implement your business logic in the client application, the impact of a change can be very painful.

Stored procedures also provide many important performance benefits. By default, a stored procedure will reuse a previously cached execution plan, saving the CPU resources and the time it takes to parse, resolve, and optimize your code. Network traffic is minimized by shortening the code strings that the client submits to Microsoft SQL Server—the client submits only the stored procedure's name and its arguments, as opposed to the full code. Moreover, all the activity is performed at the server, avoiding multiple roundtrips between the client and the server. The stored procedure passes only the final result to the client through the network.

This chapter explores stored procedures. It starts with brief coverage of the different types of stored procedures supported by SQL Server 2008 and then delves into details. The chapter covers the stored procedure's interface, resolution process, compilation, recompilations and execution plan reuse, plan guides, the EXECUTE AS clause, and common language runtime (CLR) stored procedures.

Types of Stored Procedures

SQL Server 2008 supports different types of stored procedures: user-defined, system, and extended. You can develop user-defined stored procedures with T-SQL or with the CLR. This section briefly covers the different types.

User-Defined Stored Procedures

A user-defined stored procedure is created in a user database and typically interacts with the database objects. When you invoke a user-defined stored procedure, you specify the EXEC (or EXECUTE) command and the stored procedure's schema-qualified name and arguments:

```
EXEC dbo.Proc1 <arguments>;
```

As an example, run the following code to create the *GetSortedShippers* stored procedure in the InsideTSQL2008 database:

```
USE InsideTSQL2008;

IF OBJECT_ID('dbo.GetSortedShippers', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers;
GO
-- Stored procedure GetSortedShippers
-- Returns shippers sorted by requested sort column
CREATE PROC dbo.GetSortedShippers
  @colname AS sysname = NULL
AS

DECLARE @msg AS NVARCHAR(500);

-- Input validation
IF @colname IS NULL
BEGIN
  SET @msg = N'A value must be supplied for parameter @colname.';
  RAISERROR(@msg, 16, 1);
  RETURN;
END

IF @colname NOT IN(N'shipperid', N'companyname', N'phone')
BEGIN
  SET @msg =
    N'Valid values for @colname are: '
    + N'N''shipperid'', N''companyname'', N''phone''.';
  RAISERROR(@msg, 16, 1);
  RETURN;
END

-- Return shippers sorted by requested sort column
IF @colname = N'shipperid'
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY shipperid;
ELSE IF @colname = N'companyname'
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY companyname;
ELSE IF @colname = N'phone'
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY phone;
GO
```

The stored procedure accepts a column name from the Sales.Shippers table in the InsideTSQL2008 database as input (*@colname*); after input validation, it returns the rows from the Shippers table sorted by the specified column name. Input validation here involves verifying that a column name was specified, and that the specified column name exists in the Shippers table. Later in the chapter, I will discuss the subject of parameterizing sort order in more detail; for now, I just wanted to provide a simple example of a user-defined stored procedure. Run the following code to invoke *GetSortedShippers* specifying *N'companyname'* as input:

```
EXEC dbo.GetSortedShippers @colname = N'companyname';
```

This generates the following output:

```
shipperid   companyname    phone
----------- -------------- ---------------
2           Shipper ETYNR  (425) 555-0136
1           Shipper GVSUA  (503) 555-0137
3           Shipper ZHISN  (415) 555-0138
```

You can leave out the keyword EXEC if the stored procedure is the first statement of a batch, but I recommend using it all the time. You can also omit the stored procedure's schema name (*dbo* in our case), but when you neglect to specify it, SQL Server must resolve the schema. The resolution in SQL Server 2008 occurs in the following order (adapted from SQL Server Books Online):

1. The sys schema of the current database.
2. The caller's default schema if executed in a batch or in dynamic SQL. Or, if the nonqualified procedure name appears inside the body of another procedure definition, the schema containing this other procedure is searched next.
3. The dbo schema in the current database.

   For example, suppose that you connect to the InsideTSQL2008 database and your user's default schema in InsideTSQL2008 is called Sales. You invoke the following code in a batch:

   ```
   EXEC GetSortedShippers @colname = N'companyname';
   ```

The resolution takes place in the following order:

1. Look for *GetSortedShippers* in the sys schema of InsideTSQL2008 (*sys.GetSortedShippers*). If found, execute it; if not, proceed to the next step (as in our case).
2. If invoked in a batch (as in our case) or dynamic SQL, look for *GetSortedShippers* in Sales (*Sales.GetSortedShippers*). Or, if invoked in another procedure (say, *Production.AnotherProc*), look for *GetSortedShippers* in *Production* next. If found, execute it; if not, proceed to the next step (as in our case).
3. Look for *GetSortedShippers* in the *dbo* schema (*dbo.GetSortedShippers*). If found (as in our case), execute it; if not, generate a resolution error.

As I mentioned earlier, you can use stored procedures as a security layer. You can control access to objects by granting execution permissions on stored procedures and not on underlying objects. For example, suppose that there's a database user called user1 in the InsideTSQL2008 database. You want to allow user1 to invoke the *GetSortedShippers* procedure, but you want to deny user1 direct access to the Shippers table. You can achieve this by granting the user EXECUTE permissions on the procedure, and denying SELECT (and possibly other) permissions on the table, as in:

```
DENY SELECT ON Sales.Shippers TO user1;
GRANT EXECUTE ON dbo.GetSortedShippers TO user1;
```

SQL Server allows user1 to execute the stored procedure. However, if user1 attempts to query the Shippers table directly:

```
SELECT shipperid, companyname, phone
FROM Sales.Shippers;
```

SQL Server generates the following error:

```
Msg 229, Level 14, State 5, Line 1
The SELECT permission was denied on the object 'Shippers', database 'InsideTSQL2008', schema
'Sales'.
```

This security model gives you a high level of control over the activities that users will be allowed to perform.

I'd like to point out other aspects of stored procedure programming through the *GetSortedShippers* sample procedure:

- Notice that I explicitly specified column names in the query and didn't use SELECT *. Using SELECT * is a bad practice. In the future, the table might undergo schema changes that cause your application to break. Also, if you really need only a subset of the table's columns and not all of them, the use of SELECT * prevents the optimizer from utilizing covering indexes defined on that subset of columns.
- The query is missing a filter. This is not a bad practice by itself—it's perfectly valid if you really need all rows from the table. But you might be surprised to learn that in performance-tuning projects at Solid Quality Mentors, we still find production applications that need filtered data but filter it only at the client. Such an approach introduces extreme pressure on both SQL Server and the network. Filters allow the optimizer to consider using indexes, which minimizes the I/O cost. Also, by filtering at the server, you reduce network traffic. If you need filtered data, make sure you filter it at the server; use a WHERE clause (or ON, HAVING where relevant)!
- Notice the use of a semicolon (;) to suffix statements. Although not a requirement of T-SQL for all statements, the semicolon suffix is an ANSI requirement. In SQL Server 2008, you are required to suffix some statements with a semicolon to avoid ambiguity

of your code. For example, the WITH keyword is used for different purposes—to define a CTE, to specify a table hint, and so on. SQL Server requires you to suffix the statement preceding the CTE's WITH clause to avoid ambiguity. Similarly, the MERGE keyword is used for different purposes—to specify a join hint and to start a MERGE statement. SQL Server requires you to terminate a MERGE statement with a semicolon to avoid ambiguity. Getting used to suffixing all statements with a semicolon is a good practice.

Now let's get back to the focus of this section—user-defined stored procedures.

As I mentioned earlier, to invoke a user-defined stored procedure, you specify EXEC, the schema-qualified name of the procedure, and the parameter values for the invocation if there are any. References in the stored procedure to system and user object names that are not fully qualified (that is, without the database prefix) are always resolved in the database in which the procedure was created. If you want to invoke a user-defined procedure created in another database, you must database-qualify its name. For example, if you are connected to a database called db1 and want to invoke a stored procedure called *dbo.Proc1*, which resides in db2, you would use the following code:

```
USE db1;
EXEC db2.dbo.Proc1 <arguments>;
```

Invoking a procedure from another database wouldn't change the fact that object names that are not fully qualified would be resolved in the database in which the procedure was created (db2, in this case).

If you want to invoke a remote stored procedure residing in another instance of SQL Server, you would use the fully qualified stored procedure name, including the linked server name: *server.database.schema.proc*.

When done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.GetSortedShippers', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers;
```

Special Stored Procedures

By *special* stored procedure I mean a stored procedure created with a name beginning with *sp_* in the master database. A stored procedure created in this way has a special behavior.

Important Note that Microsoft strongly recommends against creating your own stored procedures with the *sp_* prefix. This prefix is used by SQL Server to designate system stored procedures. In this section, I will create stored procedures with the *sp_* prefix to demonstrate their special behavior.

As an example, the following code creates the special procedure *sp_Proc1*, which prints the database context and queries the INFORMATION_SCHEMA.TABLES view—first with dynamic SQL, then with a static query:

```
USE master;
IF OBJECT_ID('dbo.sp_Proc1', 'P') IS NOT NULL DROP PROC dbo.sp_Proc1;
GO

CREATE PROC dbo.sp_Proc1
AS
PRINT 'master.dbo.sp_Proc1 executing in ' + DB_NAME();

-- Dynamic query
EXEC('SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = N''BASE TABLE'';');

-- Static query
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE';
GO
```

One of the unique aspects of a special procedure is that you don't need to database-qualify its name when connected to another database. For example, you can be connected to InsideTSQL2008 and still be able to run it without database-qualifying its name:

```
USE InsideTSQL2008;
EXEC dbo.sp_Proc1;
```

The PRINT command returns *master.dbo.sp_Proc1 executing in InsideTSQL2008*. The database name in the printed message was obtained by the DB_NAME function. It seems that DB_NAME "thinks" that the database context is InsideTSQL2008 (the current database) and not master. Similarly, dynamic SQL also assumes the context of the current database; therefore, the EXEC command (which invokes a query against INFORMATION_SCHEMA.TABLES) returns table names from the InsideTSQL2008 database. In contrast to the previous two statements, the static query against INFORMATION_SCHEMA.TABLES seems to "think" that it is running in master—it returns table names from the master database and not InsideTSQL2008. Similarly, if you refer with static code to user objects (for example, a table called T1), SQL Server looks for them in master. If that's not confusing enough, static code referring to compatibility views (for example, sys.objects) is normally resolved in master, but if the catalog view is a backward-compatibility one (for example, sys.sysobjects) the code is resolved in the current database.

Interestingly, the *sp_* prefix also works magic with other types of objects in addition to stored procedures.

Caution The behavior described in the following section is undocumented and you should not rely on it in production environments.

For example, the following code creates a table with the *sp_* prefix in master:

```
USE master;
IF OBJECT_ID('dbo.sp_Globals', 'U') IS NOT NULL
  DROP TABLE dbo.sp_Globals;

CREATE TABLE dbo.sp_Globals
(
  var_name sysname     NOT NULL PRIMARY KEY,
  val      SQL_VARIANT NULL
);
```

And the following code switches between database contexts and always manages to find the table even though the table name is not database-qualified.

```
USE InsideTSQL2008;
INSERT INTO dbo.sp_Globals(var_name, val)
  VALUES('var1', 10);

USE AdventureWorks2008;
INSERT INTO dbo.sp_Globals(var_name, val)
  VALUES('var2', CAST(1 AS BIT));

USE tempdb;
SELECT var_name, val FROM dbo.sp_Globals;
```

This generates the following output:

```
var_name  val
--------- ----
var1      10
var2      1
```

For cleanup, run the following code:

```
USE master;
IF OBJECT_ID('dbo.sp_Globals', 'U') IS NOT NULL
  DROP TABLE dbo.sp_Globals;
```

Do not drop *sp_Proc1* yet; we'll use it in the following section.

System Stored Procedures

System stored procedures are procedures that were shipped by Microsoft. Historically, system stored procedures resided in the master database, had the *sp_* prefix, and were marked as system objects with a special flag (MS Shipped). In SQL Server 2008, system stored procedures reside physically in an internal hidden Resource database, and they exist logically in every database.

A special procedure (*sp_* prefix, created in master) that is also marked as a system procedure gets additional unique behavior. You can mark a procedure as a system procedure by using the undocumented procedure *sp_MS_marksystemobject*.

Caution You should not use the *sp_MS_marksystemobject* stored procedure in production because you won't get any support if you run into trouble with it. Also, the behavior you get by marking your procedures as *system* isn't guaranteed to remain the same in future versions of SQL Server, or even future service packs. I'm going to use it here for demonstration purposes to show additional behaviors that system procedures have.

Run the following code to mark the special procedure *sp_Proc1* also as a system procedure:

```
USE master;
EXEC sp_MS_marksystemobject 'dbo.sp_Proc1';
```

If you now run *sp_Proc1* in databases other than master, you will observe that all code statements within the stored procedure assume the context of the current database:

```
USE InsideTSQL2008;
EXEC dbo.sp_Proc1;

USE AdventureWorks2008;
EXEC dbo.sp_Proc1;

EXEC InsideTSQL2008.dbo.sp_Proc1;
```

As a practice, avoid using the *sp_* prefix for user-defined stored procedures. Remember that if a local database has a stored procedure with the same name and schema as a special procedure in master, the user-defined procedure will be invoked. To demonstrate this, create a procedure called *sp_Proc1* in InsideTSQL2008 as well:

```
USE InsideTSQL2008;
IF OBJECT_ID('dbo.sp_Proc1', 'P') IS NOT NULL DROP PROC dbo.sp_Proc1;
GO

CREATE PROC dbo.sp_Proc1
AS
PRINT 'InsideTSQL2008.dbo.sp_Proc1 executing in ' + DB_NAME();
GO
```

If you run the following code, you will observe that when connected to InsideTSQL2008, *sp_Proc1* from InsideTSQL2008 was invoked:

```
USE InsideTSQL2008;
EXEC dbo.sp_Proc1;

USE AdventureWorks2008;
EXEC dbo.sp_Proc1;
```

Drop the InsideTSQL2008 version so that it doesn't interfere with the following examples:

```
USE InsideTSQL2008;
IF OBJECT_ID('dbo.sp_Proc1', 'P') IS NOT NULL DROP PROC dbo.sp_Proc1;
```

Interestingly, system procedures have an additional unique behavior: They also resolve user objects in the current database, not just system objects. To demonstrate this, run the following code to re-create the *sp_Proc1* special procedure—which queries a user table called Sales.Orders—and to mark the procedure as system:

```
USE master;
IF OBJECT_ID('dbo.sp_Proc1', 'P') IS NOT NULL DROP PROC dbo.sp_Proc1;
GO

CREATE PROC dbo.sp_Proc1
AS
PRINT 'master.dbo.sp_Proc1 executing in ' + DB_NAME();
SELECT orderid FROM Sales.Orders;
GO

EXEC sp_MS_marksystemobject 'dbo.sp_Proc1';
```

Run *sp_Proc1* in InsideTSQL2008, and you will observe that the query ran successfully against the Sales.Orders table in InsideTSQL2008:

```
USE InsideTSQL2008;
EXEC dbo.sp_Proc1;
```

Make a similar attempt in AdventureWorks2008:

```
USE AdventureWorks2008;
EXEC dbo.sp_Proc1;
```

You get the following error:

```
master.dbo.sp_Proc1 executing in AdventureWorks2008
Msg 208, Level 16, State 1, Procedure sp_Proc1, Line 4
Invalid object name 'Sales.Orders'.
```

The error tells you that SQL Server looked for a Sales.Orders table in AdventureWorks2008 but couldn't find one.

When you're done, run the following code for cleanup:

```
USE master;
IF OBJECT_ID('dbo.sp_Proc1', 'P') IS NOT NULL DROP PROC dbo.sp_Proc1;

USE InsideTSQL2008
IF OBJECT_ID('dbo.sp_Proc1', 'P') IS NOT NULL DROP PROC dbo.sp_Proc1;
```

Other Types of Stored Procedures

SQL Server also supports other types of stored procedures:

- **Temporary stored procedures** You can create temporary procedures by prefixing their names with a single number symbol or a double number symbol (# or ##).

A single number symbol makes the procedure a local temporary procedure; two number symbols make it a global one. Local and global temporary procedures behave in terms of visibility and scope like local and global temporary tables, respectively.

More Info For details about local and global temporary tables, please refer to Chapter 7, "Temporary Tables and Table Variables."

- **Extended stored procedures** These procedures allow you to create external routines with a programming language such as C using the Extended Stored Procedure API. These were used in older versions of SQL Server to extend the functionality of the product. External routines were written using the Extended Stored Procedure API, compiled to a .dll file, and registered as extended stored procedures in SQL Server. They were used like user-defined stored procedures with T-SQL. In SQL Server 2008, extended stored procedures are supported for backward compatibility and will be removed in a future version of SQL Server. Now you can rely on the .NET integration in the product and develop CLR stored procedures, as well as other types of routines. I'll cover CLR procedures later in the chapter.

The Stored Procedure Interface

This section covers the interface (that is, the input and output parameters) of stored procedures. Stored procedures accept three kinds of parameters: scalar input parameters, table-valued input parameters, and scalar output parameters.

Scalar Input Parameters

A scalar input parameter must be provided with a value when the stored procedure is invoked, unless you assign the parameter with a default value. For example, the following code creates the *GetCustOrders* procedure, which accepts a customer ID and datetime range boundaries as inputs, and returns the given customer's orders in the given datetime range:

```
USE InsideTSQL2008;
IF OBJECT_ID('dbo.GetCustOrders', 'P') IS NOT NULL
  DROP PROC dbo.GetCustOrders;
GO

CREATE PROC dbo.GetCustOrders
  @custid   AS INT,
  @fromdate AS DATETIME = '19000101',
  @todate   AS DATETIME = '99991231'
AS

SET NOCOUNT ON;
```

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid
  AND orderdate >= @fromdate
  AND orderdate < @todate;
GO
```

Tip The SET NOCOUNT ON option tells SQL Server not to produce the message saying how many rows were affected for data manipulation language (DML) statements. Some client database interfaces, such as OLEDB, absorb this message as a row set. The result is that when you expect to get a result set of a query back to the client, you instead get this message of how many rows were affected as the first result set. By issuing SET NOCOUNT ON, you avoid this problem in those interfaces, so you might want to adopt the practice of specifying it.

When invoking a stored procedure, you must specify inputs for those scalar parameters that were not given default values in the definition (for *@custid* in our case). There are two formats for assigning values to parameters when invoking a stored procedure: *unnamed* and *named*. In the unnamed format, you just specify values without specifying the parameter names. Also, you must specify the inputs by declaration order of the parameters. You can omit inputs only for parameters that have default values and that were declared at the end of the parameter list. You cannot omit an input between two parameters for which you do specify values. If you want such parameters to use their default values, you need to specify the DEFAULT keyword for those.

As an example, the following code invokes the procedure without specifying the inputs for the two last parameters, which will use their default values:

```
EXEC dbo.GetCustOrders 1;
```

This generates the following output:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- -----------------------
10692       1           4           2007-10-03 00:00:00.000
10702       1           4           2007-10-13 00:00:00.000
10643       1           6           2007-08-25 00:00:00.000
10835       1           1           2008-01-15 00:00:00.000
10952       1           1           2008-03-16 00:00:00.000
11011       1           3           2008-04-09 00:00:00.000
```

If you want to specify your own value for the third parameter but use the default for the second, specify the DEFAULT keyword for the second parameter:

```
EXEC dbo.GetCustOrders 1, DEFAULT, '20100212';
```

And, of course, if you want to specify your own values for all parameters, just specify them in order:

```
EXEC dbo.GetCustOrders 1, '20070101', '20080101';
```

This produces the following output:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- -----------------------
10643       1           6           2007-08-25 00:00:00.000
10692       1           4           2007-10-03 00:00:00.000
10702       1           4           2007-10-13 00:00:00.000
```

These are the basics of stored procedures. You're probably already familiar with them, but I decided to include this coverage to lead to a recommended practice. Many maintenance-related issues can arise when you use the unnamed assignment format. You must specify the arguments in order; you must not omit an optional parameter; and by looking at the code, it might not be clear what the inputs actually mean and to which parameter they relate. Therefore, it's a good practice to use the named assignment format, in which you specify the name of the argument and assign it with an input value, as in the following example:

```
EXEC dbo.GetCustOrders
  @custid   = 1,
  @fromdate = '20070101',
  @todate   = '20080101';
```

The code is much more readable, you can play with the order in which you specify the inputs, and you can omit any parameter that you like if it has a default value.

Table-Valued Parameters

SQL Server 2008 introduces table types and table-valued parameters. A table type allows you to store the definition of a table structure as a user-defined object in the database and later use it as the type for table variables and table-valued parameters. A table type definition can include most common elements of a table definition, including column names; types; NULLability; properties such as IDENTITY and COLLATE; and constraint definitions such as PRIMARY KEY, UNIQUE, and CHECK (but not FOREIGN KEY).

For example, the following code creates a table type called dbo.OrderIDs in the InsideTSQL2008 database:

```
USE InsideTSQL2008;

IF TYPE_ID('dbo.OrderIDs') IS NOT NULL DROP TYPE dbo.OrderIDs;

CREATE TYPE dbo.OrderIDs AS TABLE
(
  pos INT NOT NULL PRIMARY KEY,
  orderid INT NOT NULL UNIQUE
);
```

The type defines orders with attributes called *pos* and *orderid* representing a position for sorting purposes and an order ID, respectively. Once created, you can use OrderIDs as the type for table variables , like so:

```
DECLARE @T AS dbo.OrderIDs;

INSERT INTO @T(pos, orderid)
  VALUES(1, 10248),(2, 10250),(3, 10249);

SELECT * FROM @T;
```

This generates the following output:

```
pos         orderid
----------- -----------
1           10248
3           10249
2           10250
```

Such use of table types as the type for table variables prevents you from the need to repeat the table definition. Of course, it’s not just about code brevity. The real news in supporting table types is that you can use those as types for input parameters in stored procedures and UDFs. Client APIs were also enhanced to support passing table-valued parameters to routines, so you are not restricted to using T-SQL to invoke such routines.

Note that when defining a table-valued parameter in a routine, you have to specify the attribute READONLY, indicating that you can only read from the parameter but not write to it. For now, this attribute is mandatory. I hope that in the future SQL Server will also support writable table-valued parameters.

As an example, the following code creates a stored procedure called *GetOrders* that accepts an input table-valued parameter called @T of the OrderIDs type, and returns the orders from the Sales.Orders table whose IDs appear in @T, sorted by *pos*:

```
IF OBJECT_ID('dbo.GetOrders', 'P') IS NOT NULL DROP PROC dbo.GetOrders;
GO

CREATE PROC dbo.GetOrders(@T AS dbo.OrderIDs READONLY)
AS

SELECT O.orderid, O.orderdate, O.custid, O.empid
FROM Sales.Orders AS O
  JOIN @T AS K
    ON O.orderid = K.orderid
ORDER BY K.pos;
GO
```

To invoke the procedure from T-SQL, first declare and populate a local table variable of the OrderIDs type in the calling batch, then call the procedure and pass the variable as the input parameter, like so:

```
DECLARE @Myorderids AS dbo.OrderIDs;

INSERT INTO @Myorderids(pos, orderid)
  VALUES(1, 10248),(2, 10250),(3, 10249);

EXEC dbo.GetOrders @T = @Myorderids;
```

This generates the following output:

```
orderid     orderdate               custid      empid
----------- ----------------------- ----------- -----------
10248       2006-07-04 00:00:00.000 85          5
10250       2006-07-08 00:00:00.000 34          4
10249       2006-07-05 00:00:00.000 79          6
```

The input parameter is passed by reference, meaning that SQL Server gets a pointer to the parameter, rather than internally generating a copy. This makes the use of table-valued parameters very efficient. SQL Server can also efficiently reuse a previously cached plan of our stored procedure for subsequent invocations of the procedure.

Internally, SQL Server treats table-valued parameters very much like table variables. This means that SQL Server does not maintain distribution statistics (histograms) on them. The downside of not having distribution statistics on table variables is that the optimizer can't come up with accurate selectivity estimates for filters. The upside is that you get fewer recompiles, because no refreshes of statistics would trigger plan optimality–related recompiles.

Note that unlike with scalar input parameters, SQL Server won't generate an error if you omit an input table-valued parameter when executing the procedure. In such a case, SQL Server simply uses an empty table by default. This means that if you omit such a parameter by mistake, you will have a logical bug in the code that might go unnoticed. For example, run the following code:

```
EXEC dbo.GetOrders;
```

You don't get an error—instead, the stored procedure uses an empty table by default, producing an empty set as the output:

```
orderid     orderdate               custid      empid
----------- ----------------------- ----------- -----------

(0 row(s) affected)
```

When you're done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.GetOrders', 'P') IS NOT NULL DROP PROC dbo.GetOrders;
IF TYPE_ID('dbo.OrderIDs') IS NOT NULL DROP TYPE dbo.OrderIDs;
```

Output Parameters

Output parameters allow you to return output values from a stored procedure. A change made to the output parameter within the stored procedure is reflected in the variable from the calling batch that was assigned to the output parameter. The concept is similar to a pointer in C or a *ByRef* parameter in Visual Basic.

As an example, the following code alters the definition of the *GetCustOrders* procedure, adding to it the output parameter *@numrows*:

```
USE InsideTSQL2008;
GO

ALTER PROC dbo.GetCustOrders
  @custid   AS INT,
  @fromdate AS DATETIME = '19000101',
  @todate   AS DATETIME = '99991231',
  @numrows  AS INT OUTPUT
AS

SET NOCOUNT ON;

DECLARE @err AS INT;

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid
  AND orderdate >= @fromdate
  AND orderdate < @todate;

SELECT @numrows = @@rowcount, @err = @@error;

RETURN @err;
GO
```

@numrows returns the number of rows affected by the query. Notice that the stored procedure also uses a RETURN clause to return the value of the *@@error* function after the invocation of the query.

To get the output parameter back from the stored procedure when invoking it, you need to assign it with a variable defined in the calling batch and mention the keyword OUTPUT. To get back the return status, you also need to provide a variable from the calling batch right before the procedure name and an equal sign, as in the following example:

```
DECLARE @myerr AS INT, @mynumrows AS INT;

EXEC @myerr = dbo.GetCustOrders
  @custid   = 1,
  @fromdate = '20070101',
  @todate   = '20080101',
  @numrows  = @mynumrows OUTPUT;

SELECT @myerr AS err, @mynumrows AS rc;
```

This generates the following output:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- -----------------------
10643       1           6           2007-08-25 00:00:00.000
10692       1           4           2007-10-03 00:00:00.000
10702       1           4           2007-10-13 00:00:00.000

err         rc
----------- -----------
0           3
```

The stored procedure returns the applicable orders, plus it assigns the return status *0* to *@myerr* and the number of affected rows (in this case, *3*) to the *@mynumrows* variable.

If you want to manipulate the row set returned by the stored procedure with T-SQL, you need to create a table first and use the INSERT/EXEC syntax, by running the following code:

```
IF OBJECT_ID('tempdb..#CustOrders', 'U') IS NOT NULL
  DROP TABLE #CustOrders;

CREATE TABLE #CustOrders
(
  orderid    INT      NOT NULL PRIMARY KEY,
  custid INT NOT NULL,
  empid INT      NOT NULL,
  orderdate  DATETIME NOT NULL
);
GO

DECLARE @myerr AS INT, @mynumrows AS INT;

INSERT INTO #CustOrders(orderid, custid, empid, orderdate)
  EXEC @myerr = dbo.GetCustOrders
    @custid   = 1,
    @fromdate = '20070101',
    @todate   = '20080101',
    @numrows  = @mynumrows OUTPUT;

SELECT orderid, custid, empid, orderdate
FROM #CustOrders;

SELECT @myerr AS err, @mynumrows AS rc;
```

When you're done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.GetCustOrders', 'P') IS NOT NULL
  DROP PROC dbo.GetCustOrders;
IF OBJECT_ID('tempdb..#CustOrders', 'U') IS NOT NULL
  DROP TABLE #CustOrders;
```

Resolution

When you create a stored procedure, SQL Server first parses the code to check for syntax errors. If the code passes the parsing stage successfully, SQL Server attempts to resolve the names it contains. The resolution process verifies the existence of object and column names, among other things. If the referenced objects exist, the resolution process will take place fully—that is, it also checks for the existence of the referenced column names.

If an object name exists but a column within it doesn't, the resolution process produces an error and the stored procedure is not created. However, if the object doesn't exist at all, SQL Server creates the stored procedure and defers the resolution process to run time, when the stored procedure is invoked. Of course, if a referenced object or a column is still missing when you execute the stored procedure, the code will fail. This process of postponing name resolution until run time is called *deferred name resolution*.

I'll demonstrate the resolution aspects I just described. First run the following code to make sure that the *Proc1* procedure, the *Proc2* procedure, and the table T1 do not exist within tempdb:

```
USE tempdb;
IF OBJECT_ID('dbo.Proc1', 'P') IS NOT NULL DROP PROC dbo.Proc1;
IF OBJECT_ID('dbo.Proc2', 'P') IS NOT NULL DROP PROC dbo.Proc2;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
```

Run the following code to create the stored procedure *Proc1*, which refers to a table named T1, which doesn't exist:

```
CREATE PROC dbo.Proc1
AS

SELECT col1 FROM dbo.T1;
GO
```

Because table T1 doesn't exist, resolution was deferred to run time, and the stored procedure was created successfully. If T1 does not exist when you invoke the procedure, it fails at run time. Run the following code:

```
EXEC dbo.Proc1;
```

You get the following error:

```
Msg 208, Level 16, State 1, Procedure Proc1, Line 4
Invalid object name 'dbo.T1'.
```

Next, create table T1 with a column called *col1*:

```
CREATE TABLE dbo.T1(col1 INT);
INSERT INTO dbo.T1(col1) VALUES(1);
```

Invoke the stored procedure again:

```
EXEC dbo.Proc1;
```

This time it will run successfully.

Next, attempt to create a stored procedure called *Proc2*, referring to a nonexistent column (*col2*) in the existing T1 table:

```
CREATE PROC dbo.Proc2
AS

SELECT col2 FROM dbo.T1;
GO
```

Here, the resolution process was not deferred to run time because T1 exists. The stored procedure was not created, and you got the following error:

```
Msg 207, Level 16, State 1, Procedure Proc2, Line 4
Invalid column name 'col2'.
```

When you're done, run the following code for cleanup:

```
USE tempdb;
IF OBJECT_ID('dbo.Proc1', 'P') IS NOT NULL DROP PROC dbo.Proc1;
IF OBJECT_ID('dbo.Proc2', 'P') IS NOT NULL DROP PROC dbo.Proc2;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
```

Dependency Information

Prior to SQL Server 2008, dependency information between objects was not reliable. Dependency information was recorded only if the referenced object existed when the referencing object was created. But if the referenced object didn't exist when the referencing object was created, and SQL Server ended up using deferred name resolution, object dependency simply wasn't recorded. So when asking for dependency information prior to SQL Server 2008 (by querying the *sys.sql_dependencies* or *sys.sysdepends* compatibility views, or by executing the *sp_depends* system procedure) the information that you got was incomplete and thus not reliable.

SQL Server 2008 addresses this problem by providing reliable dependency information. When you create an object, SQL Server parses its text and records dependency information regardless of whether the referenced object exists . SQL Server exposes dependency information through three objects: the *sys.sql_expression_dependencies* catalog view and the *sys.dm_sql_referenced_entities* and *sys.dm_sql_referencing_entities* dynamic management functions (DMFs). I will explain the purpose of each object shortly. When querying these objects, if a referenced object doesn't exist, you get only object name information. If a referenced object exists, you get both object name and ID information. Dependency information keeps track of all relevant parts, including server, database, schema, object, and even column.

Note that SQL Server records only dependency information for references that appear in static T-SQL code. It doesn't record dependency information for references that appear in dynamic SQL and CLR code.

To demonstrate retrieving reliable dependency information in SQL Server 2008, first run the following code, which creates a few objects with dependencies:

```
USE tempdb;

IF OBJECT_ID('dbo.Proc1', 'P') IS NOT NULL DROP PROC dbo.Proc1;
IF OBJECT_ID('dbo.Proc2', 'P') IS NOT NULL DROP PROC dbo.Proc2;
IF OBJECT_ID('dbo.V1', 'V') IS NOT NULL DROP VIEW dbo.V1;
IF OBJECT_ID('dbo.V2', 'V') IS NOT NULL DROP VIEW dbo.V2;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
IF OBJECT_ID('dbo.T2', 'U') IS NOT NULL DROP TABLE dbo.T2;
GO

CREATE PROC dbo.Proc1
AS
SELECT * FROM dbo.T1;
EXEC('SELECT * FROM dbo.T2');
GO
CREATE PROC dbo.Proc2
AS
SELECT * FROM dbo.T3;
GO
CREATE TABLE dbo.T1(col1 INT);
CREATE TABLE dbo.T2(col2 INT);
GO
CREATE VIEW dbo.V1
AS
SELECT col1 FROM dbo.T1;
GO
CREATE VIEW dbo.V2
AS
SELECT col1 FROM dbo.T1;
GO
```

Observe that the procedure *Proc1* has a dependency on the table T1 in static code and on the table T2 in dynamic code. Also notice that the procedure is created before the referenced tables are created. The procedure *Proc2* has a reference to the table T3 in static code, but the table doesn't exist. The views V1 and V2 refer to the column *col1* in T1.

Next I'll describe the purpose of the three objects that give you dependency information, or more accurately, information about references by name. The *sys.sql_expression_dependencies* view gives you object dependency information by name. Run the following code in the tempdb database:

```
SELECT
  OBJECT_SCHEMA_NAME(referencing_id) AS srcobjschema,
  OBJECT_NAME(referencing_id) AS srcobjname,
  referencing_minor_id AS srcminorid,
  referenced_schema_name AS tgtschema,
  referenced_id AS tgtobjid,
  referenced_entity_name AS tgtobjname,
  referenced_minor_id AS tgtminorid
FROM sys.sql_expression_dependencies;
```

You get the following output:

```
srcobjschema  srcobjname  srcminorid  tgtschema  tgtobjid    tgtobjname  tgtminorid
------------- ----------- ----------- ---------- ----------- ----------- -----------
dbo           Proc1       0           dbo        2098106515  T1          0
dbo           Proc2       0           dbo        NULL        T3          0
dbo           V1          0           dbo        2098106515  T1          0
dbo           V2          0           dbo        2098106515  T1          0
```

Observe that you got only dependency information for references that appear in static code. The reference in *Proc1* to T2 in the dynamic SQL code wasn't recorded. Also observe that for referenced objects that exist (for example, T1) you get both name and ID information, but for objects that don't exist (for example, T3) you get only name information. For objects that exist, you get the object ID even if the dependency was established before the object existed—the ID is effectively "filled in" when known.

If you attempt to run code that refers to a nonexistent object, you get a Level 16 resolution error. If you want to know which objects a certain object depends on, query the *sys.dm_sql_referenced_entities* function and provide the referencing object name as the first input and OBJECT as the second input, like so:

```
SELECT
  referenced_schema_name AS objschema,
  referenced_entity_name AS objname,
  referenced_minor_name  AS minorname,
  referenced_class_desc  AS class
FROM sys.dm_sql_referenced_entities('dbo.Proc1', 'OBJECT');
```

You get the following output, indicating that *Proc1* depends on the table T1 and on the column *col1* within T1:

```
objschema  objname  minorname  class
---------- -------- ---------- -----------------
dbo        T1       NULL       OBJECT_OR_COLUMN
dbo        T1       col1       OBJECT_OR_COLUMN
```

If you want to know which objects depend on a certain object—for example before dropping an object—query the *sys.dm_sql_referencing_entities* function and provide the referenced object name as the first input and 'OBJECT' as the second input, like so:

```
SELECT
  referencing_schema_name AS objschema,
  referencing_entity_name AS objname,
  referencing_class_desc  AS class
FROM sys.dm_sql_referencing_entities('dbo.T1', 'OBJECT');
```

This generates the following output indicating that *Proc1*, V1, and V2 depend on T1:

```
objschema  objname  class
---------- -------- -----------------
dbo        Proc1    OBJECT_OR_COLUMN
dbo        V1       OBJECT_OR_COLUMN
dbo        V2       OBJECT_OR_COLUMN
```

Important These objects contain a bit less than full dependency information—they give you information about references by name (with some exceptions, such as references to temporary tables). If there's an indirect dependency (e.g., *Proc3* calls *Proc1*, and *Proc1* references T1 by name), the simple calls to these objects won't reveal it (here, *Proc3* depends on T1). However, an indirect dependency like this may be just as important to know about as a direct named reference. If you want to know what will be affected when you drop an object, for example, you have to follow the dependencies yourself. The good news is that in SQL Server 2008, you can.

When you're done experimenting with object dependency information run the following code for cleanup:

```
IF OBJECT_ID('dbo.Proc1', 'P') IS NOT NULL DROP PROC dbo.Proc1;
IF OBJECT_ID('dbo.V1', 'V') IS NOT NULL DROP VIEW dbo.V1;
IF OBJECT_ID('dbo.V2', 'V') IS NOT NULL DROP VIEW dbo.V2;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
IF OBJECT_ID('dbo.T2', 'U') IS NOT NULL DROP TABLE dbo.T2;
```

Compilations, Recompilations, and Reuse of Execution Plans

Earlier I mentioned that when you create a stored procedure, SQL Server parses your code and then attempts to resolve it. If resolution was deferred, it will take place at first invocation. Upon first invocation of the stored procedure, if the resolution phase finished successfully, SQL Server analyzes and optimizes the queries within the stored procedure and generates an execution plan. An execution plan holds the instructions to process the query. These instructions include which order to access the tables in; which indexes, access methods, and join algorithms to use; whether to spool interim sets; and so on. SQL Server typically generates multiple permutations of execution plans and will choose the one with the lowest cost out of the ones that it generated.

Note that SQL Server won't necessarily create all possible permutations of execution plans; if it did, the optimization phase might take too long. SQL Server limits the optimizer by calculating a threshold for optimization based on the sizes of the tables involved as well as other factors.

Stored procedures can reuse a previously cached execution plan, thereby saving the resources involved in generating a new execution plan. This section will discuss the reuse of execution plans, cases when a plan cannot be reused, parameter and variable sniffing issues, and plan guides.

Reuse of Execution Plans

The process of optimization requires mainly CPU resources. SQL Server will, by default, reuse a previously cached plan from an earlier invocation of a stored procedure, without investigating whether it is actually a good idea to do so.

To demonstrate plan reuse, first run the following code, which creates the *GetOrders* stored procedure:

```
USE InsideTSQL2008;
IF OBJECT_ID('dbo.GetOrders', 'P') IS NOT NULL DROP PROC dbo.GetOrders;
GO

CREATE PROC dbo.GetOrders
  @odate AS DATETIME
AS

SELECT orderid, custid, empid, orderdate /* 33145F87-1109-4959-91D6-F1EC81F8428F */
FROM Sales.Orders
WHERE orderdate >= @odate;
GO
```

The stored procedure accepts an order date as input (*@odate*) and returns orders placed on or after the input order date. I embedded a comment with a GUID in the code to be able to easily track down cached plans that are associated with this query.

Turn on the STATISTICS IO option to get back I/O information for your session's activity:

```
SET STATISTICS IO ON;
```

Run the stored procedure for the first time, providing an input with *high selectivity* (that is, an input for which a small percentage of rows will be returned):

```
EXEC dbo.GetOrders '20080506';
```

This generates the following output:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- -----------------------
11074       73          7           2008-05-06 00:00:00.000
11075       68          8           2008-05-06 00:00:00.000
11076       9           4           2008-05-06 00:00:00.000
11077       65          1           2008-05-06 00:00:00.000
```

Examine the execution plan produced for the query, shown in Figure 3-1.

Because this is the first time the stored procedure is invoked, SQL Server generated an execution plan for it based on the selective input value and cached that plan.

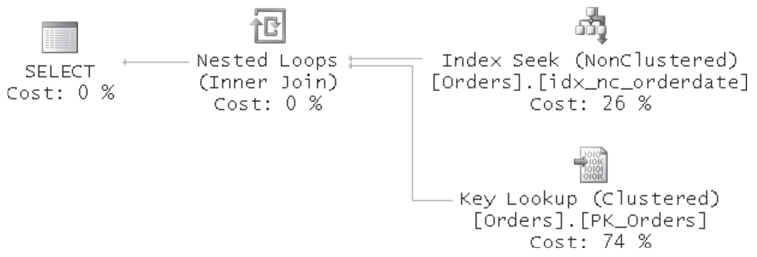


FIGURE 3-1 Execution plan showing that the index on *orderdate* is used

The optimizer uses cardinality and density information to estimate the cost of the access methods that it considers applying, and the selectivity of filters is an important factor. For example, a query with a highly selective filter can benefit from a nonclustered, noncovering index, whereas a *low selectivity* filter (that is, one that returns a high percentage of rows) would not justify using such an index.

For highly selective input such as that provided to our stored procedure, the optimizer chose a plan that uses a nonclustered, noncovering index on the *orderdate* column. The plan first performed a seek within that index (Index Seek operator), reaching the first index entry that matches the filter at the leaf level of the index. This seek operation caused two page reads, one at each of the two levels in the index. In a larger table, such an index might contain three or four levels.

Following the seek operation, the plan performed a partial ordered forward scan within the leaf level of the index (which is not seen in the plan but is part of the Index Seek operator). The partial scan fetched all index entries that match the query's filter (that is, all *orderdate* values greater than or equal to the input *@odate*). Because the input was very selective, only four matching *orderdate* values were found. In this particular case, the partial scan did not need to access additional pages at the leaf level beyond the leaf page that the seek operation reached, so it did not incur additional I/O.

The plan used a Nested Loops operator, which invoked a series of Clustered Index Seek operations to look up the data row for each of the four index entries that the partial scan found. Because the clustered index on this small table has two levels, the lookups cost eight page reads: *2 × 4 = 8*. In total, there were 10 page reads: *2 (seek) + 2 × 4 (lookups) = 10*. This is the value reported by STATISTICS IO as logical reads.

That's the optimal plan for this selective query with the existing indexes.

Remember that I mentioned earlier that stored procedures will, by default, reuse a previously cached plan. Now that you have a plan stored in cache, additional invocations of the stored procedure will reuse it. That's fine if you keep invoking the stored procedure with a highly selective input. You will enjoy the fact that the plan is reused, and SQL Server will not waste resources on generating new plans. That's especially important with systems that invoke stored procedures very frequently.

However, imagine that the stored procedure's inputs vary considerably in selectivity—some invocations have high selectivity whereas others have low selectivity. For example, the following code invokes the stored procedure with an input that has low selectivity:

```
EXEC dbo.GetOrders '20060101';
```

Because a plan is in cache, it will be reused, which is unfortunate in this case. I provided an input value earlier than the earliest *orderdate* in the table. This means that all rows in the table (830) qualify. The plan will require a clustered index lookup for each qualifying row. This invocation generated 1,664 logical reads, even though the whole Orders table resides on 21 data pages. Keep in mind that the Orders table is very small and that in production environments such a table would typically have millions of rows. The cost of reusing such a plan would then be much more dramatic given a similar scenario. For example, take a table with 1,000,000 orders residing on about 25,000 pages. Suppose that the clustered index contains three levels. Just the cost of the lookups would then be 3,000,000 reads: *1,000,000 × 3 = 3,000,000*.

Obviously, in a case such as this, with a lot of data access and large variations in selectivity, it's a very bad idea to reuse a previously cached execution plan.

Similarly, if you invoke the stored procedure for the first time with a low selectivity input, you get a plan that is optimal for that input—one that issues a table scan (unordered clustered index scan)—and that plan would be cached. Then, in later invocations, the plan would be reused even when the input has high selectivity.

You can observe the fact that an execution plan was reused by querying the *sys.syscacheobjects* view, which contains information about execution plans:

```
SELECT cacheobjtype, objtype, usecounts, sql
FROM sys.syscacheobjects
WHERE sql NOT LIKE '%sys%'
  AND sql LIKE '%33145F87-1109-4959-91D6-F1EC81F8428F%';
```

As you can see, planting a GUID in a comment embedded in the query makes it very easy to filter only the plans of interest. This code generates the following output:

```
cacheobjtype   objtype  usecounts  sql
-------------- -------- ---------- ----------------------------
Compiled Plan  Proc     2          CREATE PROC dbo.GetOrders...
```

Notice that one plan was found for the *GetOrders* procedure in cache, and that it was used twice (*usecounts* = 2).

One way to solve the problem is to create two stored procedures—one for requests with high selectivity and a second for low selectivity. You create another stored procedure with flow logic, examining the input and determining which procedure to invoke based on the input's selectivity that your calculations estimate. The idea is nice in theory, but it's very difficult to implement in practice. It can be very complex to calculate the boundary point dynamically without consuming additional resources. Furthermore, this stored procedure accepts only one input, so imagine how complex things would become with multiple inputs.

Another way to solve the problem is to create (or alter) the stored procedure with the RECOMPILE option, as in:

```
ALTER PROC dbo.GetOrders
  @odate AS DATETIME
WITH RECOMPILE
AS

SELECT orderid, custid, empid, orderdate /* 33145F87-1109-4959-91D6-F1EC81F8428F */
FROM Sales.Orders
WHERE orderdate >= @odate;
GO
```

The RECOMPILE option tells SQL Server to create a new execution plan every time it is invoked. This option actually tells SQL Server not to bother to cache the plan, hence every invocation of the procedure ends up creating a new plan because it won't find an existing one. It is especially useful when the cost of the recompiles is lower than the extra cost associated with reusing suboptimal plans.

First, run the altered procedure specifying an input with high selectivity:

```
EXEC dbo.GetOrders '20080506';
```

You get the plan shown earlier in Figure 3-1, which is optimal in this case and generates an I/O cost of 10 logical reads.

Next, run it specifying an input with low selectivity:

```
EXEC dbo.GetOrders '20060101';
```

You get the plan in Figure 3-2, showing a table scan (unordered clustered index scan), which is optimal for this input. The I/O cost in this case is 21 logical reads.

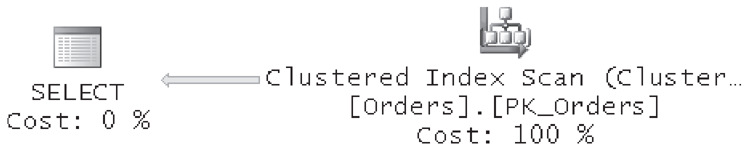


FIGURE 3-2 Execution plan showing a table scan (unordered clustered index scan)

As mentioned, when creating a stored procedure with the RECOMPILE option, SQL Server doesn't even bother to keep the execution plan for it in cache. If you now query *sys.syscacheobjects*, you will get no plan back for the *GetOrders* procedure:

```
SELECT cacheobjtype, objtype, usecounts, sql
FROM sys.syscacheobjects
WHERE sql NOT LIKE '%sys%'
  AND sql LIKE '%33145F87-1109-4959-91D6-F1EC81F8428F%';
```

If you had multiple queries in your stored procedure, the RECOMPILE procedure option would cause all of them to get recompiled every time the procedure ran. Of course, that's a waste of resources if only some of the queries would benefit from recompiles whereas others would benefit from plan reuse.

SQL Server 2008 supports statement-level recompiles. Instead of having all queries in the stored procedure recompiled, SQL Server can recompile individual statements. You can request a statement-level recompile by specifying a query hint called RECOMPILE (not to be confused with the RECOMPILE procedure option). This way, other queries can benefit from reusing previously cached execution plans if you don't have a reason to recompile them every time the stored procedure is invoked.

Run the following code to alter the procedure, specifying the RECOMPILE query hint:

```
ALTER PROC dbo.GetOrders
  @odate AS DATETIME
AS

SELECT orderid, custid, empid, orderdate /* 33145F87-1109-4959-91D6-F1EC81F8428F */
FROM Sales.Orders
WHERE orderdate >= @odate
OPTION(RECOMPILE);
GO
```

In our case, there's only one query in the stored procedure, so it doesn't really matter whether you specify the RECOMPILE option at the procedure or the query level. But try to think of the advantages of this hint when you have multiple queries in one stored procedure.

Note There is a certain difference between the RECOMPILE procedure option and the RECOMPILE query hint that might be worth noting, and it is regarding estimated execution plans. When the procedure is created with the RECOMPILE procedure option, there is no cached plan. Consequently, a new plan is generated when the estimated plan is requested. If the RECOMPILE query option is used, however, a plan is cached, as mentioned, and the RECOMPILE query option forces a recompile at run time, but not at estimated-plan-generation time. In other words, estimated and actual query plans may not match when statement-level OPTION (RECOMPILE) is present. They will match if the procedure-level RECOMPILE option is used instead.

To see that you get good plans, first run the procedure specifying an input with high selectivity:

```
EXEC dbo.GetOrders '20080506';
```

You will get the plan in Figure 3-1 and an I/O cost of 10 logical reads.

Next, run it specifying an input with low selectivity:

```
EXEC dbo.GetOrders '20060101';
```

You will get the plan in Figure 3-2 and an I/O cost of 21 logical reads.

Don't be confused by the fact that *syscacheobjects* shows a plan with the value *2* as the *usecounts*:

```
SELECT cacheobjtype, objtype, usecounts, sql
FROM sys.syscacheobjects
WHERE sql NOT LIKE '%sys%'
  AND sql LIKE '%33145F87-1109-4959-91D6-F1EC81F8428F%';
```

This generates the following output:

```
cacheobjtype   objtype  usecounts  sql
-------------- -------- ---------- ----------------------------
Compiled Plan  Proc     2          CREATE PROC dbo.GetOrders...
```

Remember that if other queries were in the stored procedure, they could potentially reuse the execution plan.

At this point, you can turn off the STATISTICS IO option:

```
SET STATISTICS IO OFF;
```

Recompilations

As I mentioned earlier, as a rule a stored procedure will reuse a previously cached execution plan by default. There are some exceptions to this rule, and in certain situations there is a recompilation even when there is a plan in cache. Remember that in SQL Server 2008, a recompilation occurs at the statement level.

Such exceptions might be caused by issues related to plan stability (correctness) or plan optimality. Plan stability issues include schema changes in underlying objects (for example, adding or dropping a column, adding or dropping an index, and so on) or changes to SET options that can affect query results (for example, ANSI_NULLS, CONCAT_NULL_YIELDS_NULL, and so on). Plan optimality issues that cause recompilation include making data changes in referenced objects to the extent that a new plan might be more optimal—for example, as a result of a statistics update.

Both types of causes for recompilations have many particular cases. At the end of this section, I will provide you with a resource that describes them in great detail.

Naturally, if a plan is removed from cache after a while for lack of reuse, SQL Server generates a new one when the procedure is invoked again.

To see an example of a cause of a recompilation, first run the following code, which creates the stored procedure *CustCities*:

```
IF OBJECT_ID('dbo.CustCities', 'P') IS NOT NULL
  DROP PROC dbo.CustCities;
GO
```

```
CREATE PROC dbo.CustCities
AS

SELECT custid, country, region, city, /* 97216686-F90E-4D5A-9A9E-CFD9E548AE81 */
  country + '.' + region + '.' + city AS CRC
FROM Sales.Customers
ORDER BY country, region, city;
GO
```

The stored procedure queries the Customers table, concatenating the three parts of the customer's geographical location: *country*, *region*, and *city*. By default, the SET option CONCAT_NULL_YIELDS_NULL is turned ON, meaning that when you concatenate a NULL with any string, you get a NULL as a result.

Run the stored procedure for the first time:

```
EXEC dbo.CustCities;
```

This generates the following output, shown here in abbreviated form:

```
custid  country    region  city            CRC
------- ---------- ------- --------------- -------------------------
12      Argentina  NULL    Buenos Aires    NULL
54      Argentina  NULL    Buenos Aires    NULL
64      Argentina  NULL    Buenos Aires    NULL
20      Austria    NULL    Graz            NULL
59      Austria    NULL    Salzburg        NULL
50      Belgium    NULL    Bruxelles       NULL
76      Belgium    NULL    Charleroi       NULL
61      Brazil     RJ      Rio de Janeiro  Brazil.RJ.Rio de Janeiro
67      Brazil     RJ      Rio de Janeiro  Brazil.RJ.Rio de Janeiro
34      Brazil     RJ      Rio de Janeiro  Brazil.RJ.Rio de Janeiro
31      Brazil     SP      Campinas        Brazil.SP.Campinas
88      Brazil     SP      Resende         Brazil.SP.Resende
81      Brazil     SP      Sao Paulo       Brazil.SP.Sao Paulo
21      Brazil     SP      Sao Paulo       Brazil.SP.Sao Paulo
15      Brazil     SP      Sao Paulo       Brazil.SP.Sao Paulo
...
```

As you can see, whenever *region* was NULL, the concatenated string became NULL. SQL Server cached the execution plan of the stored procedure for later reuse. Along with the plan, SQL Server also stored the state of all SET options that can affect query results. You can observe those in a bitmap called *setopts* in *sys.syscacheobjects*.

Set the CONCAT_NULL_YIELDS_NULL option to OFF, telling SQL Server to treat a NULL in concatenation as an empty string:

```
SET CONCAT_NULL_YIELDS_NULL OFF;
```

Rerun the stored procedure:

```
EXEC dbo.CustCities;
```

This generates the following output:

```
custid  country    region  city            CRC
------- ---------- ------- --------------- -------------------------
12      Argentina  NULL    Buenos Aires    Argentina..Buenos Aires
54      Argentina  NULL    Buenos Aires    Argentina..Buenos Aires
64      Argentina  NULL    Buenos Aires    Argentina..Buenos Aires
20      Austria    NULL    Graz            Austria..Graz
59      Austria    NULL    Salzburg        Austria..Salzburg
50      Belgium    NULL    Bruxelles       Belgium..Bruxelles
76      Belgium    NULL    Charleroi       Belgium..Charleroi
61      Brazil     RJ      Rio de Janeiro  Brazil.RJ.Rio de Janeiro
67      Brazil     RJ      Rio de Janeiro  Brazil.RJ.Rio de Janeiro
34      Brazil     RJ      Rio de Janeiro  Brazil.RJ.Rio de Janeiro
31      Brazil     SP      Campinas        Brazil.SP.Campinas
88      Brazil     SP      Resende         Brazil.SP.Resende
81      Brazil     SP      Sao Paulo       Brazil.SP.Sao Paulo
21      Brazil     SP      Sao Paulo       Brazil.SP.Sao Paulo
15      Brazil     SP      Sao Paulo       Brazil.SP.Sao Paulo
...
```

You can see that when *region* was NULL, it was treated as an empty string, and as a result, you didn't get a NULL in the *CRC* column. Changing the session option in this case changed the meaning of a query. When you ran this stored procedure, SQL Server first checked for a cached plan that also had the same state of SET options. SQL Server didn't find one, so it had to generate a new plan.

Query *sys.syscacheobjects*:

```
SELECT cacheobjtype, objtype, usecounts, setopts, sql
FROM sys.syscacheobjects
WHERE sql NOT LIKE '%sys%'
  AND sql LIKE '%97216686-F90E-4D5A-9A9E-CFD9E548AE81%';
```

In the output, you find two plans for *CustCities* with two different *setopts* bitmaps:

```
cacheobjtype   objtype  usecounts  setopts  sql
-------------- -------- ---------- -------- ------------------------------
Compiled Plan  Proc     1          4347     CREATE PROC dbo.CustCities...
Compiled Plan  Proc     1          4339     CREATE PROC dbo.CustCities...
```

Turn the CONCAT_NULL_YIELDS_NULL option back ON:

```
SET CONCAT_NULL_YIELDS_NULL ON;
```

Note that regardless of whether the change in the SET option affects the query's meaning, SQL Server looks for a match in the set options state to reuse a plan. For example, run the following code to re-create the procedure *GetOrders* that I used in my previous examples:

```
IF OBJECT_ID('dbo.GetOrders', 'P') IS NOT NULL DROP PROC dbo.GetOrders;
GO
```

```
CREATE PROC dbo.GetOrders
  @odate AS DATETIME
AS

SELECT orderid, custid, empid, orderdate /* 33145F87-1109-4959-91D6-F1EC81F8428F */
FROM Sales.Orders
WHERE orderdate >= @odate;
GO
```

Run the procedure for the first time when the set option is on:

```
EXEC dbo.GetOrders '20080506';
```

Run the following code to turn the option off:

```
SET CONCAT_NULL_YIELDS_NULL OFF;
```

Run the procedure a second time when the option is off:

```
EXEC dbo.GetOrders '20080506';
```

Inspect the cached plans associated with the query:

```
SELECT cacheobjtype, objtype, usecounts, setopts, sql
FROM sys.syscacheobjects
WHERE sql NOT LIKE '%sys%'
  AND sql LIKE '%33145F87-1109-4959-91D6-F1EC81F8428F%';
```

Observe in the output that there are two plans:

```
cacheobjtype   objtype  usecounts  setopts  sql
-------------- -------- ---------- -------- ------------------------------
Compiled Plan  Proc     1          4339     CREATE PROC dbo.GetOrders...
Compiled Plan  Proc     1          4347     CREATE PROC dbo.GetOrders...
```

No concatenation is going on in the *GetOrders* procedure, so clearly the change in this set option doesn't have any impact on the behavior of the code. Still, SQL Server just compares bitmaps and creates a new plan if they are different.

Why should you care? Client interfaces and tools typically change the state of some SET options whenever you make a new connection to the database. Different client interfaces change different sets of options, yielding different execution environments. If you're using multiple database interfaces and tools to connect to the database and they have different execution environments, they won't be able to reuse each other's plans. You can easily identify the SET options that each client tool changes by running a trace while the applications connect to the database, or by running DBCC USEROPTIONS. If you see discrepancies in the execution environment, you can code explicit SET commands in all applications, which will be submitted whenever a new connection is made. This way, all applications have sessions with the same execution environment and can reuse one another's plans.

When you're done testing run the following code to set the option back on:

```
SET CONCAT_NULL_YIELDS_NULL ON;
```

As for recompiles caused by plan optimality, a classic example is refresh of distribution statistics (histograms). After SQL Server refreshes statistics, the next time you run a query with a nontrivial plan that relies on those statistics the plan will be recompiled. SQL Server makes the assumption that data distribution may have changed to the degree that a different plan might be optimal. If you know that your procedure is called with such inputs that the query would keep benefiting from the same plan even after statistics refresh, you can specify the KEEPFIXED PLAN query hint. This hint indicates to SQL Server not to perform plan optimality–related recompiles.

This section offered just a couple of examples for recompiles. There are many others. Later I'll provide a resource where you can find more.

Variable Sniffing

As I mentioned earlier, SQL Server generates a plan for a stored procedure based on the inputs provided to it upon first invocation, for better or worse. *First invocation* also refers to the first invocation after a plan was removed from cache for lack of reuse (or for any other reason). This capability is called *parameter sniffing*, meaning that when optimizing the code, the optimizer can "sniff" the values of the procedure's parameters. The optimizer "knows" the values of the input parameters, and it generates an adequate plan for those inputs. However, things are different when you refer to local variables in your queries. And for the sake of our discussion, it doesn't matter whether these are local variables of a plain batch or of a stored procedure. When optimizing the code at the batch level, the optimizer cannot sniff the content of the variables; therefore, when it optimizes the query, it must make a guess. Obviously, this can lead to poor plans if you're not aware of the problem and don't take corrective measures.

To demonstrate the problem, first insert a new order to the Orders table, specifying the CURRENT_TIMESTAMP function for the *orderdate* column:

```
INSERT INTO Sales.Orders
  (custid, empid, orderdate, requireddate, shippeddate, shipperid, freight,
   shipname, shipaddress, shipcity, shipregion, shippostalcode, shipcountry)
 VALUES
  (1, 1, CURRENT_TIMESTAMP, '20100212 00:00:00.000', NULL, 1, 1,
   N'a', N'a', N'a', N'a', N'a', N'a');
```

Re-create the *GetOrders* stored procedure so that it declares a local variable and use it in the query's filter:

```
IF OBJECT_ID('dbo.GetOrders', 'P') IS NOT NULL DROP PROC dbo.GetOrders;
GO
```

```
CREATE PROC dbo.GetOrders
  @d AS INT = 0
AS

DECLARE @odate AS DATETIME;
SET @odate = DATEADD(day, -@d, CONVERT(VARCHAR(8), CURRENT_TIMESTAMP, 112));

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= @odate;
GO
```

The procedure defines the integer input parameter *@d* with a default value *0*. It declares a *datetime* local variable called *@odate*, which is set to today's date minus *@d* days. The stored procedure then issues a query returning all orders with an *orderdate* greater than or equal to *@odate*. Invoke the stored procedure using the default value of *@d*:

```
EXEC dbo.GetOrders;
GO
```

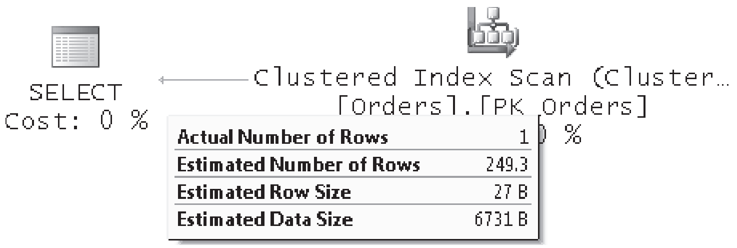
This generates the following output:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- -----------------------
11078       1           1           2009-03-09 04:19:01.540
```

Note The output that you get will have a value in *orderdate* that reflects the CURRENT_TIMESTAMP value of when you inserted the new order.

Unlike recompiles, initial compiles take place at the batch level—not the statement level. Therefore, the optimizer didn't know what the value of *@odate* was when it optimized the query. So it used a conservative, hard-coded value that is 30 percent of the number of rows in the table. For such a low-selectivity estimation, the optimizer naturally chose a full clustered index scan, even though the query in practice is highly selective and would be much better off using the index on *orderdate*.

You can observe the optimizer's estimation and chosen plan by looking at the execution plan. The actual execution plan you get for this invocation of the stored procedure is shown in Figure 3-3.



Actual Number of Rows	1
Estimated Number of Rows	249.3
Estimated Row Size	27 B
Estimated Data Size	6731 B

FIGURE 3-3 Execution plan showing estimated number of rows

You can see that the optimizer chose a table scan (unordered clustered index scan) because of its selectivity estimation of 30 percent (249.3 rows/831 total number of rows), although in actuality only one row was returned.

You can tackle this problem in several ways. One is to use, whenever possible, inline expressions in the query that refer to the input parameter instead of a variable. In our case, it is possible:

```
ALTER PROC dbo.GetOrders
  @d AS INT = 0
AS

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= DATEADD(day, -@d, CONVERT(VARCHAR(8), CURRENT_TIMESTAMP, 112));
GO
```

Run *GetOrders* again, and notice the use of the index on *orderdate* in the execution plan:

```
EXEC dbo.GetOrders;
```

The plan that you get is similar to the one shown earlier in Figure 3-1. The I/O cost here is just four logical reads.

Another way to deal with the problem is to use a stub procedure—that is, create two procedures. The first procedure accepts the original parameter, assigns the result of the calculation to a local variable, and invokes a second procedure providing it with the variable as input. The second procedure accepts an input order date passed to it and invokes the query that refers directly to the input parameter. When a plan is generated for the procedure that actually invokes the query (the second procedure), the value of the parameter will, in fact, be known at optimization time.

Run the following code to implement this solution:

```
IF OBJECT_ID('dbo.GetOrdersQuery', 'P') IS NOT NULL
  DROP PROC dbo.GetOrdersQuery;
GO

CREATE PROC dbo.GetOrdersQuery
  @odate AS DATETIME
AS

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= @odate;
GO

ALTER PROC dbo.GetOrders
  @d AS INT = 0
AS
```

```
DECLARE @odate AS DATETIME;
SET @odate = DATEADD(day, -@d, CONVERT(VARCHAR(8), CURRENT_TIMESTAMP, 112));

EXEC dbo.GetOrdersQuery @odate;
GO
```

Invoke the *GetOrders* procedure:

```
EXEC dbo.GetOrders;
```

You get an optimal plan for the input similar to the one shown earlier in Figure 3-1, yielding an I/O cost of only four logical reads.

Don't forget the issues I described in the previous section regarding the reuse of execution plans. The fact that you got an efficient execution plan for this input doesn't necessarily mean that you would want to reuse it in following invocations. It all depends on whether the inputs are typical or atypical. Make sure you follow the recommendations I gave earlier in case the inputs are atypical.

The stub procedure approach is a bit convoluted, however. SQL Server supports two much simpler options to tackle the variable sniffing problem—using the OPTIMIZE FOR and RECOMPILE query hints. Which of the two you use depends on whether you want to cache and reuse the plan. If you want to reuse the plan, use the OPTIMIZE FOR hint. This hint allows you to provide SQL Server with a literal that reflects the selectivity of the variable, in case the input is typical. For example, if you know that the variable will typically end up with a highly selective value, as you did in our example, you can provide the literal *'99991231'*, which reflects that:

```
ALTER PROC dbo.GetOrders
  @d AS INT = 0
AS

DECLARE @odate AS DATETIME;
SET @odate = DATEADD(day, -@d, CONVERT(VARCHAR(8), CURRENT_TIMESTAMP, 112));

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= @odate
OPTION(OPTIMIZE FOR(@odate = '99991231'));
GO
```

Run the stored procedure:

```
EXEC dbo.GetOrders;
```

You get an optimal plan for a highly selective *orderdate* similar to the one shown earlier in Figure 3-1, yielding an I/O cost of four logical reads.

If, on the other hand, you don't want to reuse the plan because the variable sometimes ends up with a selective value and sometimes a nonselective one, the OPTIMIZE FOR hint won't help you. Surprisingly, in such a case, the RECOMPILE query hint that I introduced earlier also resolves the variable sniffing problem. If you think about it, the variable sniffing problem has to do with SQL Server's default choice of compiling the whole batch as a unit initially. Recompiles, on the other hand, happen at the statement level. By specifying the RECOMPILE query hint (as opposed to the RECOMPILE procedure option), you explicitly request the compilation of this query to happen at the statement level. The benefit in this approach is that by the time SQL Server gets to optimize the query, the preceding statements—including the assignment of the variable—were already executed; hence the value of the variable is known at this stage. So at the cost of recompiling the statement in every invocation of the procedure, the optimization is aware of the variable's value, and this usually results in more efficient plans.

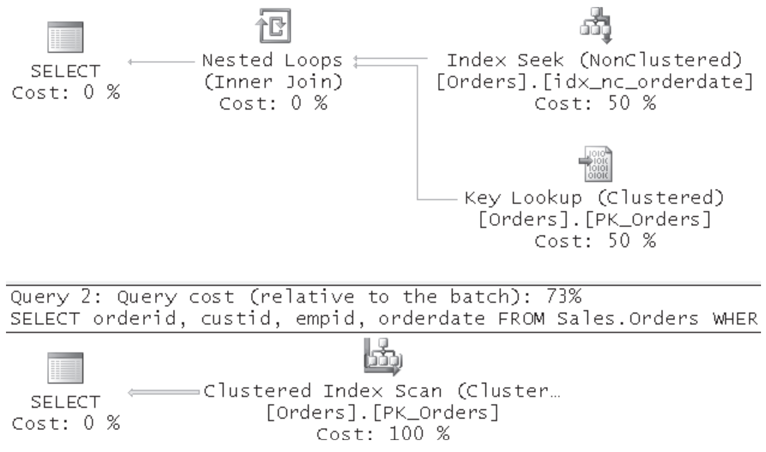
Run the following code to re-create the stored procedure with the RECOMPILE query option:

```
ALTER PROC dbo.GetOrders
  @d AS INT = 0
AS

DECLARE @odate AS DATETIME;
SET @odate = DATEADD(day, -@d, CONVERT(VARCHAR(8), CURRENT_TIMESTAMP, 112));

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= @odate
OPTION(RECOMPILE);
GO
```

Run the procedure with selective and nonselective values:

```
EXEC dbo.GetOrders @d = 1;
EXEC dbo.GetOrders @d = 365;
```

Examine the execution plans for the two invocations shown in Figure 3-4, and observe that both got optimal plans—the first for a selective filter and the second for a nonselective one.



```
Query 2: Query cost (relative to the batch): 73%
SELECT orderid, custid, empid, orderdate FROM Sales.Orders WHER
```

FIGURE 3-4 Execution plans for selective and nonselective filters

Note that you might face similar problems to variable sniffing when changing the values of input parameters before using them in queries. For example, say you define an input parameter called *@odate* and assign it with a default value of NULL. Before using the parameter in the query's filter, you apply the following code:

```
SET @odate = COALESCE(@odate, '19000101');
```

The query then filters orders where *orderdate* >= *@odate*. If optimization happens at the batch level, when the query is optimized the optimizer is not aware of the fact that *@odate* has undergone a change, and it optimizes the query with the original input (NULL) in mind. You will face a similar problem to the one I described with variables, and you should tackle it using similar logic.

Ironically, you may end up facing kind of an inverse problem to the variable sniffing one with parameters. As mentioned and demonstrated earlier, SQL Server does support parameter sniffing.

OPTIMIZE FOR UNKNOWN

In the previous section I described scenarios in which the optimizer doesn't sniff variable values or the correct parameter values and ends up generating an execution plan based on atypical inputs. I explained that if you knew that plan reuse was a good thing for the query, and you also knew which static values the query should be optimized for, you could use the OPTIMIZE FOR hint and provide those static values as the variable or parameter values.

Another scenario that you might face is one in which the majority of the invocations of your procedure are with typical inputs, but occasionally the procedure is invoked with atypical ones. You do know that the query would benefit from plan reuse provided that the cached plan would be the one that was optimized for the typical inputs. But the risk is that upon first invocation of the procedure—after service restart, recompile, or any other reason—the procedure will be invoked with atypical inputs, and you will end up with a cached plan that is suboptimal for the typical invocation of the stored procedure. If you have static input values that adequately represent the common case—both currently and in the future—you can use the OPTIMIZE FOR hint and specify those values. But what if there are no such static values?

SQL Server 2008 enhances the OPTIMIZE FOR hint, allowing you to indicate that you want the optimizer to optimize the query for unknown variable or parameter input values. With this option you tell SQL Server to use its existing algorithms based on statistical data to optimize the query, as opposed to attempting to sniff the inputs. For example, recall from earlier discussions that I mentioned that for a range filter the optimizer uses a selectivity estimate of 30 percent when it cannot sniff the input. By using the OPTIMIZE FOR UNKNOWN hint, you tell the optimizer that you actually want it to use such estimates even in scenarios where it could technically sniff the inputs.

You can indicate that specific parameters or variables should be assumed as unknown for optimization by using the following form:

```
<query> OPTION(OPTIMIZER FOR(@p1 UNKNOWN, @p2 UNKNWON, ...);
```

You can also indicate that all parameters and variables in the query should be assumed as unknown for optimization by using the following form:

```
<query> OPTION(OPTIMIZER FOR UNKNWON);
```

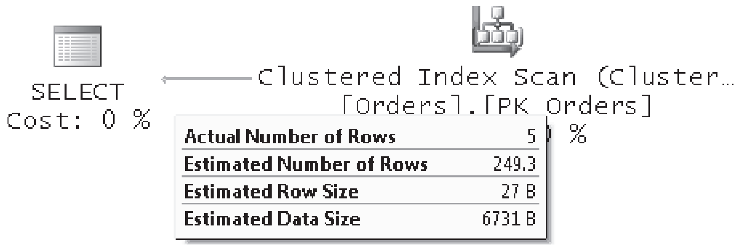
To demonstrate using this hint, run the following code, re-creating the stored procedure *GetOrders*, indicating to the optimizer to optimize the query as if the value of the *@odate* parameter is unknown:

```
IF OBJECT_ID('dbo.GetOrders', 'P') IS NOT NULL DROP PROC dbo.GetOrders;
GO

CREATE PROC dbo.GetOrders
  @odate AS DATETIME
AS

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= @odate
OPTION(OPTIMIZE FOR (@odate UNKNOWN));
GO
```

Suppose that in the vast majority of the invocations of the procedure, the inputs have low selectivity, but the procedure is invoked first time with a highly selective input, as follows:

```
EXEC dbo.GetOrders @odate = '20080506';
```

The execution plan for this execution is shown in Figure 3-5.



Actual Number of Rows	5
Estimated Number of Rows	249.3
Estimated Row Size	27 B
Estimated Data Size	6731 B

FIGURE 3-5 Execution plan for query with OPTIMIZE FOR UNKNOWN hint

Observe that you got a plan that is optimal for a low selectivity filter even though the input was highly selective. Also notice that the estimated number of rows shows 249.3, which is 30 percent of the number of rows in the table. That's the value that the optimizer optimized the query for. The actual number of rows returned was 5. This plan was cached, so the common subsequent invocations of the procedure that specify inputs with low selectivity can benefit from reusing this plan.

Plan Guides

SQL Server allows you to specify hints in your queries to force certain behavior. The three categories of hints are table, join, and query. Query hints are specified in an OPTION clause at the end of the query and indicate certain behavior at the whole query, or statement, level. Earlier in the chapter I covered a few query hints: OPTIMIZE FOR, RECOMPILE, and KEEPFIXED PLAN. SQL Server supports many others, and you can find details about those in SQL Server Books Online.

Before proceeding I should also mention the usual disclaimer regarding hints. You should use them with care, because a hint forces SQL Server to behave in a certain way, overriding its default behavior in that respect. Especially with performance hints such as forcing certain join ordering (FORCE ORDER), certain join algorithm ({LOOP | MERGE | HASH} JOIN), and so on, you force that part of optimization to become static. With respect to the part of optimization that you forced, you prevent dynamic cost-based optimization that would have normally taken place, and that would have taken into consideration data distribution and other changes.

Assuming that you know what you're doing and that you have your reasons to use a query hint, adding such a hint requires you to change the query's code. You need to add the OPTION clause with the relevant hint. However, changing a query's code is not always an option. For example, the code might be submitted from a third-party application, or your service-level agreements may prevent you from making revisions even if you technically could make them.

SQL Server 2008 supports a feature called *plan guides* that allows you to attach a query hint to a query without changing the query's code. A plan guide is an object in the database; as such, as soon as it is created and until it is dropped or disabled, it affects the query it is associated with.

You create a plan guide by using the *sp_create_plan_guide* stored procedure. You drop or disable a plan guide by using the *sp_control_plan_guide* procedure. SQL Server supports three types of plan guides, and you indicate the type of plan you want to create using the *sp_create_plan_guide* procedure under the *@type* parameter, whose valid values are N'OBJECT', N'SQL', and N'TEMPLATE'. You should use the type OBJECT when the statement appears in the context of a T-SQL routine such as a stored procedure. You should use the type SQL when the statement appears in the context of a stand-alone statement or batch. Finally, use the type TEMPLATE when you want to override the database's parameterization behavior for a certain class of statements. You need ALTER permission on the referenced object to create an object plan guide, and ALTER permissions on the database to create a SQL or template plan guide.

The next sections provide details about the three plan guide types.

Object Plan Guides

Use an object plan guide when the statement that you want to apply the hint to resides in a T-SQL routine. The applicable routines are: stored procedures, scalar UDFs, multi-statement table-valued UDFs, and DML triggers in the current database.

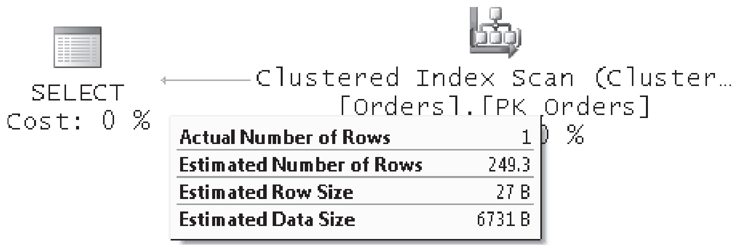
To demonstrate object plan guides, first run the following code, which re-creates the stored procedure *GetOrders* that you used in the section “Variable Sniffing” earlier:

```
IF OBJECT_ID('dbo.GetOrders', 'P') IS NOT NULL DROP PROC dbo.GetOrders;
GO

CREATE PROC dbo.GetOrders
  @d AS INT = 0
AS

DECLARE @odate AS DATETIME;
SET @odate = DATEADD(day, -@d, CONVERT(VARCHAR(8), CURRENT_TIMESTAMP, 112));

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= @odate;
GO
```

Remember that if there’s no plan in cache, the next time the procedure is invoked SQL Server will optimize the code at the batch level. Therefore, during optimization of the query the value of the variable *@odate* will be unknown. Now invoke the procedure relying on the default value of the input parameter *@d*:

```
EXEC dbo.GetOrders;
```

The query’s filter ends up being very selective, but because the optimizer wasn’t aware of the value of *@odate* the optimization was for an unknown value of *@odate* (30 percent selectivity for a range filter). You get the execution plan shown in Figure 3-6, showing a full unordered clustered index scan.



Actual Number of Rows	1
Estimated Number of Rows	249.3
Estimated Row Size	27 B
Estimated Data Size	6731 B

FIGURE 3-6 Execution plan without object plan guide

Suppose that you know that the value of *@odate* usually ends up being very selective, and you want to add the query hint *OPTION (OPTIMIZE FOR (@odate = ‘99991231’))* to the procedure's query. However, you can't—or aren't allowed to—change the procedure's code directly. You create a plan guide for the procedure's query by running the following code:

```
EXEC sp_create_plan_guide
  @name = N'PG_GetOrders_Selective',
  @stmt = N'SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= @odate;',
  @type = N'OBJECT',
  @module_or_batch = N'dbo.GetOrders',
  @hints = N'OPTION (OPTIMIZE FOR (@odate = ''99991231''))';
```

You specify the plan guide name in the *@name* argument; the statement to which you want to add the hint in the *@stmt* argument; the type OBJECT in the *@type* argument; the name of the procedure in the *@module_or_batch* argument; and finally the hint itself in the *@hints* argument.

Note Regarding object plan guides, SQL Server doesn't expect an exact match between the query in the procedure and the one you specify in the *@stmt* argument. For example, SQL Server doesn't expect an exact match in terms of use of white spaces. If SQL Server can't match the statement in the *@stmt* argument with a statement in the procedure, it generates an error such as the following:

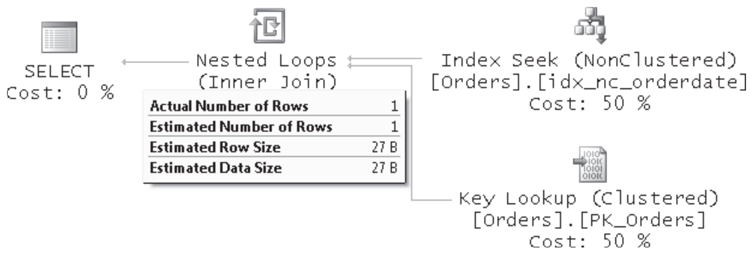
```
Msg 10507, Level 16, State 1, Procedure GetOrders, Line 2
Cannot create plan guide 'PG_GetOrders_Selective' because the statement specified by
@stmt and @module_or_batch, or by @plan_handle and @statement_start_offset, does not
match any statement in the specified module or batch. Modify the values to match a
statement in the module or batch.
```

So if you don't get an error, this fact by itself is a kind of validation that SQL Server successfully managed to match your plan guide with a statement in the procedure. Unfortunately, with the other types of plan guides SQL Server is much more strict, expecting an exact match between the code that you provide when creating the guide and the code that executes against SQL Server that is supposed to use the guide.

Execute the stored procedure again:

```
EXEC dbo.GetOrders;
```

You get the execution plan shown in Figure 3-7, showing that the index on the *orderdate* column was used, as is optimal for a selective filter.



Actual Number of Rows	1
Estimated Number of Rows	1
Estimated Row Size	27 B
Estimated Data Size	27 B

FIGURE 3-7 Execution plan with object plan guide

If you want to know whether the plan guide was used, examine the XML form of the query's execution plan. For object and SQL plan guides you should find the attributes *PlanGuideDB* and *PlanGuideName*, which are self-explanatory. You can obtain the XML plan form of a query using the SHOWPLAN_XML or STATISTICS XML set options, or the *Showplan XML* trace event. For example, run the following code to return the XML form of the query's estimated execution plan:

```
SET SHOWPLAN_XML ON;
GO
EXEC dbo.GetOrders;
GO
SET SHOWPLAN_XML OFF;
```

Within the XML plan you will find these attributes: *PlanGuideDB="InsideTSQL2008"* and *PlanGuideName="PG_GetOrders_Selective"*.

You can also query the *sys.plan_guides* view to get information about the plan guides that are stored in the current database, like so:

```
SELECT * FROM sys.plan_guides;
```

When you're done, run the following code to drop the plan guide *PG_GetOrders_Selective*:

```
EXEC sp_control_plan_guide N'DROP', N'PG_GetOrders_Selective';
```

Note that if you only want to disable the plan guide but still keep it in the database for later enabling, you should specify DISABLE instead of DROP when invoking the *sp_control_plan_guide* procedure. Later you can call the *sp_control_plan_guide* procedure again with the ENABLE option to enable the plan guide.

SQL Plan Guides

SQL plan guides are guides for statements that are submitted in the context of a stand-alone statement or batch. The statement can be submitted through any mechanism.

When creating the plan guide using the *sp_create_plan_guide* procedure, specify N'SQL' in the *@type* argument. The way you use some of the other arguments depends on the scenario. If you want the plan guide to be used only when the statement specified in the *@stmt* argument appears in the context of some batch containing multiple statements, specify the batch in the *@module_or_batch* argument. If you specify NULL for the *@module_or_batch* argument, SQL Server internally sets it to the value of *@stmt*.

Note Unlike object plan guides, for SQL plan guides SQL Server requires an exact match between the text specified when creating the plan guide and the text used when submitting the code. This applies both to the batch specified in the *@module_or_batch* argument and to the statement specified in the *@stmt* argument.

The argument *@params* is applicable when the statement is parameterized. If the statement is not parameterized, specify a NULL in this argument or omit it since NULL is the default. As an example for a plan guide for a nonparameterized stand-alone query, suppose that you want to attach a plan guide to the following query, restricting its maximum degree of parallelism to 1:

```
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

Run the following code to create the plan guide:

```
EXEC sp_create_plan_guide
  @name = N'PG_MyQuery1_MAXDOP1',
  @stmt = N'SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
',
  @type = N'SQL',
  @module_or_batch = NULL,
  @hints = N'OPTION (MAXDOP 1)';
```

As you can see, the *@module_or_batch* argument was set to NULL because this statement is not part of a batch with multiple statements, and the *@params* argument was set to NULL because the statement is not parameterized.

To verify that the plan guide was used, request the XML form of the execution plan by running the following code:

```
SET SHOWPLAN_XML ON;
GO
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
GO
SET SHOWPLAN_XML OFF;
```

Make sure that you find the attributes *PlanGuideDB="InsideTSQL2008"* and *PlanGuideName= "PG_MyQuery1_MAXDOP1"*.

To request information about the plan guide, issue the following query:

```
SELECT *
FROM sys.plan_guides
WHERE name = 'PG_MyQuery1_MAXDOP1';
```

When you're done, run the following code to drop the plan guide:

```
EXEC sp_control_plan_guide N'DROP', N'PG_MyQuery1_MAXDOP1';
```

If the statement is parameterized either explicitly (for example, when submitted through *sp_executesql*), or implicitly internally by SQL Server, specify the parameterized form in the *@stmt* argument and the parameters declaration in the *@params* argument. For statements that are parameterized using *sp_executesql*, specify the exact form used in the *@stmt* and *@ params* argument of *sp_executesql* in the corresponding *@stmt* and *@params* arguments of *sp_create_plan_guide*. For statements that get parameterized internally by SQL Server, use the *sp_get_query_template* procedure to create their parameterized form.

In the next section I'll demonstrate how to create a SQL plan guide for a parameterized query.

Template Plan Guides

Use template plan guides when you want to override the database's parameterization behavior for a certain class of statements. The database's parameterization behavior can be set to either SIMPLE (the default) or FORCED, using the database option PARAMETERIZATION. Simple parameterization means that for simple cases SQL Server internally tries to substitute constants with arguments to increase the chances for reusing previous cached plans for the same class of queries, even when the referenced constants are different. With simple parameterization, only a small class of query types are parameterized. Using forced parameterization, you increase the chances for queries to get parameterized; with few exceptions, all constants will be substituted with arguments during compilation.

Using a template plan guide you can override the database's parameterization behavior for a specified class of queries.

Because SQL Server is very strict about how it matches the text of the plan guide to the text of the query submitted to SQL Server, you should use the *sp_get_query_template* procedure to produce the text for both the query template and the parameters declaration. You specify the text of a sample query as the input parameter of the *sp_get_query_template* procedure, and you get the text of a normalized form of the query template and the parameters declaration through the procedure's output parameters. As an example, the following code demonstrates using the *sp_get_query_template* procedure to generate the text for the query template and parameters declaration for a given query:

```
DECLARE @stmt AS NVARCHAR(MAX);
DECLARE @params AS NVARCHAR(MAX);

EXEC sp_get_query_template
  @querytext = N'SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= ''99991231'';',
  @templatetext  = @stmt OUTPUT,
  @parameters  = @params OUTPUT;

SELECT @stmt AS stmt, @params AS params;
```

This generates the following output:

```
stmt
---------------------------------------------------------------------------------------
select orderid , custid , empid , orderdate from Sales . Orders where orderdate > = @0
params
-----------------
@0 varchar(8000)
```

To create a template plan guide for a certain class of queries, run the *sp_create_plan_guide* procedure, specifying N'TEMPLATE' in the *@type* argument, NULL in the *@module_or_batch* argument, the query template in the *@stmt* argument, and the parameters declaration text in the *@params* argument.

Note that you can combine template and SQL plan guides. For example, for a certain class of queries you can override the database's parameterization behavior by creating a template plan guide. For the same class of queries you can create a SQL plan guide to add any query hint that you want. The following is an example of creating both a template plan guide and a SQL plan guide for the query template shown at the beginning of this section:

```
-- Create template plan guide to use forced parameterization
DECLARE @stmt AS NVARCHAR(MAX);
DECLARE @params AS NVARCHAR(MAX);

EXEC sp_get_query_template
  @querytext = N'SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= ''99991231'';',
  @templatetext  = @stmt OUTPUT,
  @parameters  = @params OUTPUT;

EXEC sp_create_plan_guide
  @name = N'PG_MyQuery2_ParameterizationForced',
  @stmt = @stmt,
  @type = N'TEMPLATE',
  @module_or_batch = NULL,
  @params = @params,
  @hints = N'OPTION(PARAMETERIZATION FORCED)';

-- Create a SQL plan guide on the query template
EXEC sp_create_plan_guide
  @name = N'PG_MyQuery2_Selective',
  @stmt = @stmt,
  @type = N'SQL',
  @module_or_batch = NULL,
  @params = @params,
  @hints = N'OPTION(OPTIMIZE FOR (@0 = ''99991231''))';
```

The template plan guide overrides parameterization behavior for our query template to forced, and the SQL plan guide adds the OPTIMIZE FOR hint to the same query template, ensuring that it will get a plan for a selective filter.

To determine whether both plan guides are used for our query template, run the following code:

```
SET SHOWPLAN_XML ON;
GO
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20060101';
GO
SET SHOWPLAN_XML OFF;
```

You should find the following attributes in the XML plan: *TemplatePlanGuideDB="Inside TSQL2008"*, *TemplatePlanGuideName="PG_MyQuery2_ParameterizationForced"*, and *PlanGuideDB="InsideTSQL2008" PlanGuideName="PG_MyQuery2_Selective"*.

Run the following query to request information about the plan guides:

```
SELECT *
FROM sys.plan_guides
WHERE name IN('PG_MyQuery2_ParameterizationForced',
'PG_MyQuery2_Selective');
```

When you're done, run the following code to drop the plan guides:

```
EXEC sp_control_plan_guide N'DROP', N'PG_MyQuery2_ParameterizationForced';
EXEC sp_control_plan_guide N'DROP', N'PG_MyQuery2_Selective';
```

Using a Fixed XML Plan

SQL Server supports a query hint called USE PLAN that you can think of as the ultimate hint. With this hint you specify an XML value representing a complete query execution plan. SQL Server also supports creating a plan guide in which you specify an XML value representing a query execution plan as the hint. You can produce the XML form of the query plan you want in a controlled environment, and then use that XML value when creating the plan guide.

As an example, run the following code to create the stored procedure *GetOrders*, which accepts an order date as input (*@odate*) and returns all orders placed on or after the input date:

```
IF OBJECT_ID('dbo.GetOrders', 'P') IS NOT NULL DROP PROC dbo.GetOrders;
GO

CREATE PROC dbo.GetOrders
  @odate AS DATETIME
AS

SELECT orderid, custid, empid, orderdate
/* 33145F87-1109-4959-91D6-F1EC81F8428F */
FROM Sales.Orders
WHERE orderdate >= @odate;
GO
```

Once again, I specified a GUID in a comment to make it easy to track down the cached plan associated with this query.

Suppose that this procedure is usually invoked with a selective filter and you want to create a plan guide that ensures that the query uses the best plan for a selective filter. I already showed several ways to achieve this—for example, by using the OPTIMIZE FOR hint. Here I'll show an example using an XML plan representation.

First, run the procedure in a controlled environment, providing a selective value as input. Next, pull the XML form of the plan from cache by querying the dynamic management objects *sys.dm_exec_query_stats*, *sys.dm_exec_sql_text*, and *sys.dm_exec_query_plan*. Finally, create the plan guide using the *sp_create_plan_guide* procedure and specify the XML plan that you pulled from cache in the *@hint* argument. Here's the complete code to achieve this task:

```
EXEC dbo.GetOrders '99991231';
GO
DECLARE @query_plan AS NVARCHAR(MAX);
SET @query_plan = CAST(
  (SELECT query_plan
   FROM sys.dm_exec_query_stats AS QS
     CROSS APPLY sys.dm_exec_sql_text(QS.sql_handle) AS ST
     CROSS APPLY sys.dm_exec_query_plan(QS.plan_handle) AS QP
   WHERE
     SUBSTRING(ST.text, (QS.statement_start_offset/2) + 1,
       ((CASE statement_end_offset
          WHEN -1 THEN DATALENGTH(ST.text)
          ELSE QS.statement_end_offset END
              - QS.statement_start_offset)/2) + 1
            ) LIKE N'%SELECT orderid, custid, empid, orderdate
/* 33145F87-1109-4959-91D6-F1EC81F8428F */
FROM Sales.Orders
WHERE orderdate >= @odate;%'
     AND ST.text NOT LIKE '%sys%') AS NVARCHAR(MAX));

EXEC sp_create_plan_guide
  @name = N'PG_GetOrders_Selective',
  @stmt = N'SELECT orderid, custid, empid, orderdate
/* 33145F87-1109-4959-91D6-F1EC81F8428F */
FROM Sales.Orders
WHERE orderdate >= @odate;',
  @type = N'OBJECT',
  @module_or_batch = N'dbo.GetOrders',
  @hints = @query_plan;
```

I used the GUID that I planted in the code to easily identify the plan associated with my specific query, but of course, you might not have such a GUID planted in your procedure's code. You can specify any part of the query that is sufficient to identify it uniquely.

Run the following code to ensure that the plan guide is used:

```
SET SHOWPLAN_XML ON;
GO
EXEC dbo.GetOrders '20080506';
GO
SET SHOWPLAN_XML OFF;
```

You should get the following attributes: *PlanGuideDB="InsideTSQL2008"* and *PlanGuideName="PG_GetOrders_Selective"*.

Run the following code to get information about the plan guide:

```
SELECT *
FROM sys.plan_guides
WHERE name = 'PG_GetOrders_Selective';
```

SQL Server provides a table-valued function called *fn_validate_plan_guide* that validates a plan guide. You may want to validate a plan guide after a schema change in a referenced object, for example. The function accepts a plan guide ID as input, which you can obtain from the *sys.plan_guides* view. If the plan guide is valid, the function returns an empty result set; otherwise, it returns the first error that it encounters.

For example, the following code drops the index on the *orderdate* column from the Sales.Orders table and then validates the plan guide PG_GetOrders_Selective:

```
BEGIN TRAN
  DROP INDEX Sales.Orders.idx_nc_orderdate;

  SELECT plan_guide_id, msgnum, severity, state, message
  FROM sys.plan_guides
    CROSS APPLY fn_validate_plan_guide(plan_guide_id)
  WHERE name = 'PG_GetOrders_Selective';
ROLLBACK TRAN
```

Because the plan guide is invalid after the index is dropped you get the following output indicating the error:

```
plan_guide_id msgnum  severity state
------------- ------- -------- ------
65544         8712    16       0

message
---------------------------------------------------------------------
Index 'InsideTSQL2008.Sales.Orders.idx_nc_orderdate',
specified in the USE PLAN hint, does not exist.
Specify an existing index, or create an index with the specified name.
```

When you're done, run the following code to drop the plan guide:

```
EXEC sp_control_plan_guide N'DROP', N'PG_GetOrders_Selective';
```

Plan Freezing

Prior to SQL Server 2008, if you wanted to create a plan guide based on an existing plan in cache you had to first pull the complete XML plan from cache and then specify it as the hint when creating the guide. SQL Server 2008 introduces a new feature called *plan freezing* to allow you to create a plan guide directly from a plan in cache by using a stored procedure called *sp_create_plan_guide_from_handle*. The procedure accepts as arguments the name you want to assign to the plan (*@name*), the plan handle (*@plan_handle*), and the statement start offset in the parent batch (*@statement_start_offset*). Of course, you can query the dynamic management objects *sys.dm_exec_query_stats*, *sys.dm_exec_sql_text*, and *sys.dm_exec_query_plan* to pull the plan handle and statement start offset for a given query. Here's an example of plan freezing:

```
EXEC dbo.GetOrders '99991231';
GO
DECLARE @plan_handle AS VARBINARY(64), @offset AS INT, @rc AS INT;

SELECT @plan_handle = plan_handle, @offset = statement_start_offset
FROM sys.dm_exec_query_stats AS QS
  CROSS APPLY sys.dm_exec_sql_text(QS.sql_handle) AS ST
  CROSS APPLY sys.dm_exec_query_plan(QS.plan_handle) AS QP
WHERE
  SUBSTRING(ST.text, (QS.statement_start_offset/2) + 1,
    ((CASE statement_end_offset
       WHEN -1 THEN DATALENGTH(ST.text)
       ELSE QS.statement_end_offset END
           - QS.statement_start_offset)/2) + 1
         ) LIKE N'%SELECT orderid, custid, empid, orderdate
/* 33145F87-1109-4959-91D6-F1EC81F8428F */
FROM Sales.Orders
WHERE orderdate >= @odate;%'
  AND ST.text NOT LIKE '%sys%';

SET @rc = @@ROWCOUNT;

IF @rc = 1
  EXEC sp_create_plan_guide_from_handle
      @name =  N'PG_GetOrders_Selective',
      @plan_handle = @plan_handle,
      @statement_start_offset = @offset;
ELSE
  RAISERROR(
    'Number of matching plans should be 1 but is %d. Plan guide not created.',
    16, 1, @rc);
```

The code executes the stored procedure *GetOrders* in a controlled environment using a selective input. The code then pulls the plan handle and statement start offset for the procedure's query from cache by querying the aforementioned objects. The code finally executes the *sp_create_plan_guide_from_handle* procedure to create a plan guide for the query.

Run the following code to check whether the plan guide is used:

```
SET SHOWPLAN_XML ON;
GO
EXEC dbo.GetOrders '20080506';
GO
SET SHOWPLAN_XML OFF;
```

You should get the following attributes in the output: *PlanGuideDB="InsideTSQL2008"* and *PlanGuideName="PG_GetOrders_Selective"*.

Run the following query to get information about the plan guide:

```
SELECT *
FROM sys.plan_guides
WHERE name = 'PG_GetOrders_Selective';
```

When you're done experimenting with this plan guide, run the following code to drop it:

```
EXEC sp_control_plan_guide N'DROP', N'PG_GetOrders_Selective';
```

When you're done experimenting with compilations, recompilations, and reuse of execution plans, run the following code for cleanup:

```
DELETE FROM Sales.Orders WHERE orderid > 11077;
DBCC CHECKIDENT('Sales.Orders', RESEED, 11077);

IF OBJECT_ID('dbo.GetOrders') IS NOT NULL
  DROP PROC dbo.GetOrders;
IF OBJECT_ID('dbo.CustCities') IS NOT NULL
  DROP PROC dbo.CustCities;
IF OBJECT_ID('dbo.GetOrdersQuery') IS NOT NULL
  DROP PROC dbo.GetOrdersQuery;
```

More Info For more information on the subject, please refer to the white paper "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005," by Arun Marathe, revised by Shu Scott, which can be accessed at *http://technet.microsoft.com/en-us/library/cc966425.aspx*. Though written originally for SQL Server 2005, most of the coverage describing SQL Server 2005 behavior is applicable to SQL Server 2008 as well.

EXECUTE AS

Stored procedures can play an important security role. You can grant users EXECUTE permissions on the stored procedure without granting them direct access to the underlying objects, thereby giving you more control over resource access. However, certain exceptions require the caller to have direct permissions on underlying objects. To avoid requiring direct permissions from the caller, all of the following must be true:

- The stored procedure and the underlying objects belong to the same owner.
- The activity is static (as opposed to using dynamic SQL).
- The activity is DML (SELECT, INSERT, UPDATE, DELETE, or MERGE), or it is an execution of another stored procedure.

If any listed item is not true, the caller will be required to have direct permissions against the underlying objects. Otherwise, the statements in the stored procedure that do not meet the requirements will fail on a security violation.

That's the default behavior in SQL Server 2008, which cannot be changed. However, you can set the security context of the stored procedure to that of another user, as if the other user were running the stored procedure. When you create the stored procedure, you can specify an EXECUTE AS clause with one of the following options:

- **CALLER (default)** Security context of the caller
- **SELF** Security context of the user creating or altering the stored procedure
- **OWNER** Security context of the owner of the stored procedure
- **'user_name'** Security context of the specified user name

Remember, all chaining rules and requirements not to have direct permissions for underlying objects still apply, but they apply to the effective user, not the calling user (unless CALLER was specified, of course).

In addition, a user that has impersonation rights can issue an independent EXECUTE AS <option> command to impersonate another entity (login or user). If this is done, it's as if the session changes its security context to that of the impersonated entity.

Parameterizing Sort Order

To practice what you've learned so far, try to provide a solution to the following task: write a stored procedure called *GetSortedShippers* that accepts a column name from the Shippers table in the InsideTSQL2008 database as one of the inputs (*@colname*), and that returns the rows from the table sorted by the input column name. Assume also that you have a sort direction as input (*@sortdir*), with the value *'A'* representing ascending order and *'D'* representing descending order. The stored procedure should be written with performance in mind—that is, it should use indexes when appropriate (for example, a clustered or nonclustered covering index on the sort column).

Here's the first suggested solution for the task:

```
USE InsideTSQL2008;
IF OBJECT_ID('dbo.GetSortedShippers', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers;
GO

CREATE PROC dbo.GetSortedShippers
  @colname AS sysname, @sortdir AS CHAR(1) = 'A'
AS
```

```
IF @sortdir = 'A'
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY
    CASE @colname
      WHEN N'shipperid'   THEN shipperid
      WHEN N'companyname' THEN companyname
      WHEN N'phone'       THEN phone
      ELSE CAST(NULL AS SQL_VARIANT)
    END
ELSE
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY
    CASE @colname
      WHEN N'shipperid'   THEN shipperid
      WHEN N'companyname' THEN companyname
      WHEN N'phone'       THEN phone
      ELSE CAST(NULL AS SQL_VARIANT)
    END DESC;
GO
```

The solution uses an IF statement to determine which of two queries to run based on the requested sort direction. The only difference between the queries is that one uses an ascending order for the sort expression and the other uses a descending order. Each query uses a single CASE expression that returns the appropriate column value based on the input column name.

Note SQL Server determines the data type of the result of a CASE expression based on the data type with the highest precedence among the possible result values of the expression—not by the data type of the actual returned value. This means, for example, that if the CASE expression returns a VARCHAR(30) value in one of the THEN clauses and an INT value in another, the result of the expression will always be INT, because INT is higher in precedence than VARCHAR. If in practice the VARCHAR(30) value is returned, SQL Server will attempt to convert it. If the value is not convertible, you get a run-time error. If it is convertible, it becomes an INT, and of course might have a different sort behavior than the original value.

To avoid issues resulting from implicit type conversion in the CASE expression, I caused an implicit conversion of all possible return values to SQL_VARIANT by specifying an expression of an SQL_VARIANT type in the ELSE clause. SQL_VARIANT has a higher precedence than all other types; therefore, SQL Server sets the data type of the CASE expression to SQL_VARIANT, but it preserves the original base types within that SQL_VARIANT.

Run the following code to test the solution, requesting to sort the shippers by *shipperid* in descending order:

```
EXEC dbo.GetSortedShippers @colname = N'shipperid', @sortdir = N'D';
```

This generates the following output:

```
shipperid   companyname    phone
----------- -------------- ---------------
3           Shipper ZHISN  (415) 555-0138
2           Shipper ETYNR  (425) 555-0136
1           Shipper GVSUA  (503) 555-0137
```

The output is logically correct, but notice the plan generated for the stored procedure, shown in Figure 3-8.

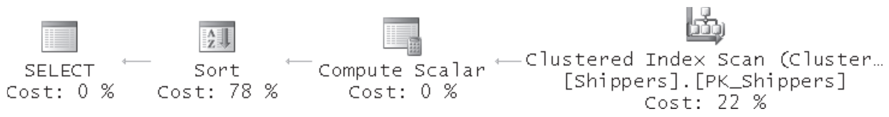


FIGURE 3-8 Execution plan showing a table scan (unordered clustered index scan) and a sort operator

Remember that the optimizer cannot rely on the sort that the index maintains if you performed manipulation on the sort column. The plan shows an unordered clustered index scan followed by an explicit sort operation. For the problem the query was intended to solve, an optimal plan would have performed an ordered scan operation in the clustered index defined on the *shipperid* column—eliminating the need for an explicit sort operation.

Here's the second solution for the task:

```
ALTER PROC dbo.GetSortedShippers
  @colname AS sysname, @sortdir AS CHAR(1) = 'A'
AS

SELECT shipperid, companyname, phone
FROM Sales.Shippers
ORDER BY
  CASE WHEN @colname = N'shipperid'   AND @sortdir = 'A'
    THEN shipperid   END,
  CASE WHEN @colname = N'companyname' AND @sortdir = 'A'
    THEN companyname END,
  CASE WHEN @colname = N'phone'       AND @sortdir = 'A'
    THEN phone       END,
  CASE WHEN @colname = N'shipperid'   AND @sortdir = 'D'
    THEN shipperid   END DESC,
  CASE WHEN @colname = N'companyname' AND @sortdir = 'D'
    THEN companyname END DESC,
  CASE WHEN @colname = N'phone'       AND @sortdir = 'D'
    THEN phone       END DESC;
GO
```

This solution uses CASE expressions in a more sophisticated way. Each column and sort direction combination is treated with its own CASE expression. Only one of the CASE expressions yields TRUE for all rows, given the column name and sort direction that particular CASE expression is looking for. All other CASE expressions return NULL for all rows. This means that only one of the CASE expressions—the one that looks for the given column name and sort direction—affects the order of the output.

Run the following code to test the stored procedure:

```
EXEC dbo.GetSortedShippers @colname = N'shipperid', @sortdir = N'D';
```

Although this stored procedure applies an interesting logical manipulation, it doesn't change the fact that you perform manipulation on the column and don't sort by it as is. This means that you will get a similar nonoptimal plan to the one shown earlier in Figure 3-8.

Here's the third solution for the task:

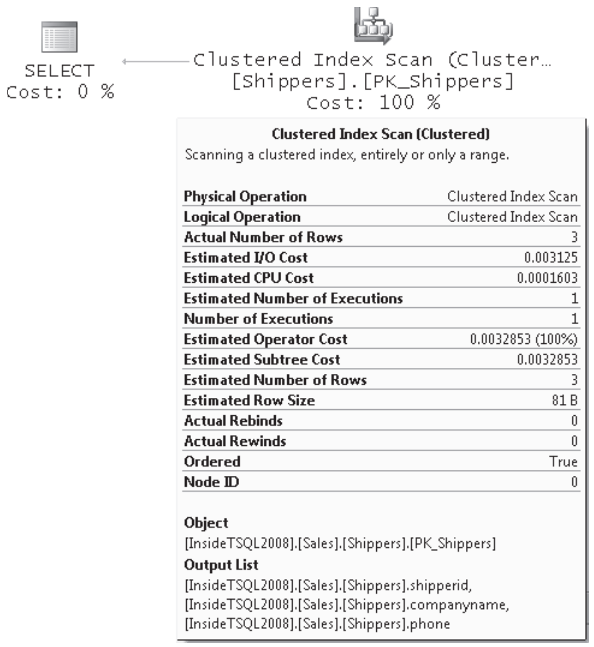
```
ALTER PROC dbo.GetSortedShippers
  @colname AS sysname, @sortdir AS CHAR(1) = 'A'
AS

IF @colname NOT IN (N'shipperid', N'companyname', N'phone')
BEGIN
  RAISERROR('Possible SQL injection attempt.', 16, 1);
  RETURN;
END

DECLARE @sql AS NVARCHAR(500);

SET @sql = N'SELECT shipperid, companyname, phone
FROM Sales.Shippers
ORDER BY '
  + QUOTENAME(@colname)
  + CASE @sortdir WHEN 'D' THEN N' DESC' ELSE '' END
  + ';';

EXEC sp_executesql @sql;
GO
```

This solution simply uses dynamic SQL concatenating the input column name and sort direction to the ORDER BY clause of the query. In terms of performance the solution achieves our goal—namely, it uses an index efficiently if an appropriate one exists. To see that it does, run the following code:

```
EXEC dbo.GetSortedShippers @colname = N'shipperid', @sortdir = N'D';
```

Observe in the execution plan, shown in Figure 3-9, that the plan performs an ordered backward clustered index scan with no sort operator, which is optimal for these inputs.

Another advantage of this solution is that it's easy to maintain. The downside of this solution is that there are some negative implications to using dynamic SQL. As mentioned earlier in this chapter, SQL Server doesn't report dependency information for code invoked with dynamic SQL. Also, dynamic SQL involves security-related issues (for example, ownership chaining and SQL injection if the inputs are not validated). For details about security and other issues related to dynamic SQL, please refer to Chapter 9, "Dynamic SQL."



Clustered Index Scan (Clustered)
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	3
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001603
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.0032853 (100%)
Estimated Subtree Cost	0.0032853
Estimated Number of Rows	3
Estimated Row Size	81 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0

Object
[InsideTSQL2008].[Sales].[Shippers].[PK_Shippers]
Output List
[InsideTSQL2008].[Sales].[Shippers].shipperid,
[InsideTSQL2008].[Sales].[Shippers].companyname,
[InsideTSQL2008].[Sales].[Shippers].phone

FIGURE 3-9 Execution plan showing an ordered backward clustered index scan

Here's the fourth solution that I'll cover:

```
CREATE PROC dbo.GetSortedShippers_shipperid_A
AS
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY shipperid;
GO
CREATE PROC dbo.GetSortedShippers_companyname_A
AS
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY companyname;
GO
CREATE PROC dbo.GetSortedShippers_phone_A
AS
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY phone;
GO
CREATE PROC dbo.GetSortedShippers_shipperid_D
AS
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY shipperid   DESC;
GO
CREATE PROC dbo.GetSortedShippers_companyname_D
AS
```

```
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY companyname DESC;
GO
CREATE PROC dbo.GetSortedShippers_phone_D
AS
  SELECT shipperid, companyname, phone
  FROM Sales.Shippers
  ORDER BY phone       DESC;
GO

ALTER PROC dbo.GetSortedShippers
  @colname AS sysname, @sortdir AS CHAR(1) = 'A'
AS

IF @colname = N'shipperid'        AND @sortdir = 'A'
  EXEC dbo.GetSortedShippers_shipperid_A;
ELSE IF @colname = N'companyname' AND @sortdir = 'A'
  EXEC dbo.GetSortedShippers_companyname_A;
ELSE IF @colname = N'phone'       AND @sortdir = 'A'
  EXEC dbo.GetSortedShippers_phone_A;
ELSE IF @colname = N'shipperid'   AND @sortdir = 'D'
  EXEC dbo.GetSortedShippers_shipperid_D;
ELSE IF @colname = N'companyname' AND @sortdir = 'D'
  EXEC dbo.GetSortedShippers_companyname_D;
ELSE IF @colname = N'phone'       AND @sortdir = 'D'
  EXEC dbo.GetSortedShippers_phone_D;
GO
```

This solution might seem awkward at first glance. You create a separate stored procedure with a single static query for each possible combination of inputs. Then, *GetSortedShippers* can act as a redirector. Simply use a series of IF/ELSE IF statements to check for each possible combination of inputs, and you explicitly invoke the appropriate stored procedure for each. Sure, it is a bit long and requires more maintenance than the previous solution, but it uses static queries that generate optimal plans. Note that each query gets its own plan and can reuse a previously cached plan for the same query.

To test the procedure, run the following code:

```
EXEC dbo.GetSortedShippers @colname = N'shipperid', @sortdir = N'D';
```

You get the optimal plan for the given inputs, similar to the plan shown earlier in Figure 3-9.

When you're done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.GetSortedShippers', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers;
IF OBJECT_ID('dbo.GetSortedShippers_shipperid_A', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers_shipperid_A;
IF OBJECT_ID('dbo.GetSortedShippers_companyname_A', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers_companyname_A;
IF OBJECT_ID('dbo.GetSortedShippers_phone_A', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers_phone_A;
```

```
IF OBJECT_ID('dbo.GetSortedShippers_shipperid_D', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers_shipperid_D;
IF OBJECT_ID('dbo.GetSortedShippers_companyname_D', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers_companyname_D;
IF OBJECT_ID('dbo.GetSortedShippers_phone_D', 'P') IS NOT NULL
  DROP PROC dbo.GetSortedShippers_phone_D;
```

CLR Stored Procedures

SQL Server 2008 allows you to develop CLR stored procedures (as well as other routines) using a .NET language of your choice. The previous chapter provided background about CLR routines, gave advice on when to develop CLR routines versus T-SQL ones, and described the technicalities of how to develop CLR routines. Remember to read Appendix A for instructions on developing, building, deploying, and testing your .NET code. Here I'd just like to give a couple of examples of CLR stored procedures that apply functionality outside the reach of T-SQL code.

The first example is a CLR procedure called *GetEnvInfo*. This stored procedure collects information from environment variables and returns it in table format. The environment variables that this procedure will return include: *Machine Name*, *Processors*, *OS Version*, *CLR Version*.

Note that to collect information from environment variables, the assembly needs external access to operating system resources. By default, assemblies are created (using the CREATE ASSEMBLY command) with the most restrictive PERMISSION_SET option, SAFE, which means that they're limited to accessing database resources only. This is the recommended option to obtain maximum security and stability. The permission set options EXTERNAL_ACCESS and UNSAFE (specified in the CREATE ASSEMBLY or ALTER ASSEMBLY commands, or in the Project | Properties dialog box in Visual Studio under the Database tab) allow external access to system resources such as files, the network, environment variables, or the registry. To allow EXTERNAL_ACCESS and UNSAFE assemblies to run, you also need to set the database option TRUSTWORTHY to ON. Allowing EXTERNAL_ACCESS or UNSAFE assemblies to run represents a security risk and should be avoided. I will describe a safer alternative shortly, but first I'll demonstrate this option. To set the TRUSTWORTHY option of the CLRUtilities database to ON and to change the permission set of the CLRUtilities assembly to EXTERNAL_ACCESS, run the following code:

```
USE CLRUtilities;

-- Database option TRUSTWORTHY needs to be ON for EXTERNAL_ACCESS
ALTER DATABASE CLRUtilities SET TRUSTWORTHY ON;

-- Alter assembly with PERMISSION_SET = EXTERNAL_ACCESS
ALTER ASSEMBLY CLRUtilities
WITH PERMISSION_SET = EXTERNAL_ACCESS;
```

At this point you can run the *GetEnvInfo* stored procedure. Keep in mind, however, that UNSAFE assemblies have complete freedom and can compromise the robustness of SQL Server and the security of the system. EXTERNAL_ACCESS assemblies get the same reliability and stability protection as SAFE assemblies, but from a security perspective they're like UNSAFE assemblies.

A more secure alternative is to sign the assembly with a strong name key file or Authenticode with a certificate. This strong name (or certificate) is created inside SQL Server as an asymmetric key (or certificate) and has a corresponding login with EXTERNAL ACCESS ASSEMBLY permission (for external access assemblies) or UNSAFE ASSEMBLY permission (for unsafe assemblies). For example, suppose that you have code in the CLRUtilities assembly that needs to run with the EXTERNAL_ACCESS permission set. You can sign the assembly with a strong-named key file from the Project | Properties dialog box in Visual Studio under the *Signing* tab. Then run the following code to create an asymmetric key from the executable .dll file and a corresponding login with the EXTERNAL_ACCESS ASSEMBLY permission:

```
-- Create an asymmetric key from the signed assembly
-- Note: you have to sign the assembly using a strong name key file
USE master

CREATE ASYMMETRIC KEY CLRUtilitiesKey
  FROM EXECUTABLE FILE =
    'C:\CLRUtilities\CLRUtilities\bin\Debug\CLRUtilities.dll'

-- Create login and grant it with external access permission
CREATE LOGIN CLRUtilitiesLogin FROM ASYMMETRIC KEY CLRUtilitiesKey
GRANT EXTERNAL ACCESS ASSEMBLY TO CLRUtilitiesLogin
```

For more details about securing your assemblies, please refer to SQL Server Books Online and to the following URL: *http://msdn2.microsoft.com/en-us/library/ms345106.aspx*.

Here's the definition of the *GetEnvInfo* stored procedure using C# code:

```
    // GetEnvInfo Procedure
    // Returns environment info in tabular format
    [SqlProcedure]
    public static void GetEnvInfo()
    {
        // Create a record - object representation of a row
        // Include the metadata for the SQL table
        SqlDataRecord record = new SqlDataRecord(
            new SqlMetaData("EnvProperty", SqlDbType.NVarChar, 20),
            new SqlMetaData("Value", SqlDbType.NVarChar, 256));
        // Marks the beginning of the result set to be sent back to the client
        // The record parameter is used to construct the metadata
        // for the result set
        SqlContext.Pipe.SendResultsStart(record);
        // Populate some records and send them through the pipe
        record.SetSqlString(0, @"Machine Name");
        record.SetSqlString(1, Environment.MachineName);
        SqlContext.Pipe.SendResultsRow(record);
        record.SetSqlString(0, @"Processors");
        record.SetSqlString(1, Environment.ProcessorCount.ToString());
        SqlContext.Pipe.SendResultsRow(record);
        record.SetSqlString(0, @"OS Version");
        record.SetSqlString(1, Environment.OSVersion.ToString());
        SqlContext.Pipe.SendResultsRow(record);
        record.SetSqlString(0, @"CLR Version");
        record.SetSqlString(1, Environment.Version.ToString());
        SqlContext.Pipe.SendResultsRow(record);
        // End of result set
        SqlContext.Pipe.SendResultsEnd();
    }
```

In this procedure, you can see the usage of some specific extensions to ADO.NET for usage within SQL Server CLR routines. These are defined in the *Microsoft.SqlServer.Server* namespace in the .NET Framework.

When you call a stored procedure from SQL Server, you are already connected. You don't have to open a new connection; you need access to the caller's context from the code running in the server. The caller's context is abstracted in a *SqlContext* object. Before using the *SqlContext* object, you should test whether it is available by using its *IsAvailable* property.

The procedure retrieves some environmental data from the operating system. The data can be retrieved by the properties of an *Environment* object, which can be found in the *System* namespace. But the data you get is in text format. In the CLR procedure, you can see how to generate a row set for any possible format. The routine's code stores data in a *SqlDataRecord* object, which represents a single row of data. It defines the schema for this single row by using the *SqlMetaData* objects.

SELECT statements in a T-SQL stored procedure send the results to the connected caller's "pipe." This is the most effective way of sending results to the caller. The same technique is exposed to CLR routines running in SQL Server. Results can be sent to the connected pipe using the send methods of the *SqlPipe* object. You can instantiate the *SqlPipe* object with the *Pipe* property of the *SqlContext* object.

Here's the definition of the *GetEnvInfo* stored procedure using Visual Basic code:

```
    ' GetEnvInfo Procedure
    ' Returns environment info in tabular format
    <SqlProcedure()> _
    Public Shared Sub GetEnvInfo()
        ' Create a record - object representation of a row
        ' Include the metadata for the SQL table
        Dim record As New SqlDataRecord( _
            New SqlMetaData("EnvProperty", SqlDbType.NVarChar, 20), _
            New SqlMetaData("Value", SqlDbType.NVarChar, 256))
        ' Marks the beginning of the result set to be sent back to the client
        ' The record parameter is used to construct the metadata for
        ' the result set
        SqlContext.Pipe.SendResultsStart(record)
        '' Populate some records and send them through the pipe
        record.SetSqlString(0, "Machine Name")
        record.SetSqlString(1, Environment.MachineName)
        SqlContext.Pipe.SendResultsRow(record)
        record.SetSqlString(0, "Processors")
        record.SetSqlString(1, Environment.ProcessorCount.ToString())
        SqlContext.Pipe.SendResultsRow(record)
        record.SetSqlString(0, "OS Version")
        record.SetSqlString(1, Environment.OSVersion.ToString())
        SqlContext.Pipe.SendResultsRow(record)
        record.SetSqlString(0, "CLR Version")
        record.SetSqlString(1, Environment.Version.ToString())
        SqlContext.Pipe.SendResultsRow(record)
        ' End of result set
        SqlContext.Pipe.SendResultsEnd()
    End Sub
```

Run the following code to register the C# version of the *GetEnvInfo* stored procedure in the CLRUtilities database:

```
USE CLRUtilities;
IF OBJECT_ID('dbo.GetEnvInfo', 'PC') IS NOT NULL
  DROP PROC dbo.GetEnvInfo;
GO
CREATE PROCEDURE dbo.GetEnvInfo
AS EXTERNAL NAME CLRUtilities.CLRUtilities.GetEnvInfo;
```

Use the following code to register the stored procedure in case you used Visual Basic to develop it:

```
CREATE PROCEDURE dbo.GetEnvInfo
AS EXTERNAL NAME
  CLRUtilities.[CLRUtilities.CLRUtilities].GetEnvInfo;
```

Run the following code to test the *GetEnvInfo* procedure:

```
EXEC dbo.GetEnvInfo;
```

This generated the following output on my computer:

```
EnvProperty          Value
-------------------- ----------------------------------------------
Machine Name         DOJO
Processors           2
OS Version           Microsoft Windows NT 6.0.6001 Service Pack 1
CLR Version          2.0.50727.3074
```

The second example for a CLR procedure creates the *GetAssemblyInfo* stored procedure, which returns information about an input assembly.

Here's the definition of the *GetAssemblyInfo* stored procedure using C# code:

```
// GetAssemblyInfo Procedure
// Returns assembly info, uses Reflection
[SqlProcedure]
public static void GetAssemblyInfo(SqlString asmName)
{
    // Retrieve the clr name of the assembly
    String clrName = null;
    // Get the context
    using (SqlConnection connection =
             new SqlConnection("Context connection = true"))
    {
        connection.Open();
        using (SqlCommand command = new SqlCommand())
        {
            // Get the assembly and load it
            command.Connection = connection;
            command.CommandText =
              "SELECT clr_name FROM sys.assemblies WHERE name = @asmName";
            command.Parameters.Add("@asmName", SqlDbType.NVarChar);
            command.Parameters[0].Value = asmName;
            clrName = (String)command.ExecuteScalar();
            if (clrName == null)
            {
                throw new ArgumentException("Invalid assembly name!");
            }
            Assembly myAsm = Assembly.Load(clrName);
            // Create a record - object representation of a row
            // Include the metadata for the SQL table
            SqlDataRecord record = new SqlDataRecord(
                new SqlMetaData("Type", SqlDbType.NVarChar, 50),
                new SqlMetaData("Name", SqlDbType.NVarChar, 256));
            // Marks the beginning of the result set to be sent back
            // to the client
            // The record parameter is used to construct the metadata
            // for the result set
            SqlContext.Pipe.SendResultsStart(record);
            // Get all types in the assembly
            Type[] typesArr = myAsm.GetTypes();
            foreach (Type t in typesArr)
            {
                // Type in a SQL database should be a class or
                // a structure
                if (t.IsClass == true)
                {
                    record.SetSqlString(0, @"Class");
                }
                else
                {
                    record.SetSqlString(0, @"Structure");
                }
                record.SetSqlString(1, t.FullName);
                SqlContext.Pipe.SendResultsRow(record);
                // Find all public static methods
```

```
                    MethodInfo[] miArr = t.GetMethods();
                    foreach (MethodInfo mi in miArr)
                    {
                        if (mi.IsPublic && mi.IsStatic)
                        {
                            record.SetSqlString(0, @"  Method");
                            record.SetSqlString(1, mi.Name);
                            SqlContext.Pipe.SendResultsRow(record);
                        }
                    }
                }
                // End of result set
                SqlContext.Pipe.SendResultsEnd();
            }
        }
    }
```

A DBA could have a problem finding out exactly what part of a particular .NET assembly is loaded to the database. Fortunately, this problem can be easily mitigated. All .NET assemblies include metadata, describing all types (classes and structures) defined within it, including all public methods and properties of the types. In .NET, the *System.Reflection* namespace contains classes and interfaces that provide a managed view of loaded types.

For a very detailed overview of a .NET assembly stored in the file system, you can use the Reflector for .NET, a very sophisticated tool created by Lutz Roeder. Because it is downloadable for free from his site at *http://www.aisto.com/roeder/dotnet/*, it is very popular among .NET developers. Also, in his blog at *http://blogs.msdn.com/sqlclr/archive/2005/11/21/495438.aspx*, Miles Trochesset wrote a SQL Server CLR DDL trigger that is fired on the CREATE ASSEMBLY statement. The trigger automatically registers all CLR objects from the assembly, including UDTs, UDAs, UDFs, SPs, and triggers. I used both tools as a starting point to create my simplified version of a SQL Server CLR stored procedure. I thought that a DBA might prefer to read the assembly metadata from a stored procedure, not from an external tool (which Lutz Roeder's Reflector for .NET is). I also thought that a DBA might just want to read the metadata first, not immediately register all CLR objects from the assembly the way that Miles Trochesset's trigger does.

The *GetAssemblyInfo* procedure has to load an assembly from the *sys.assemblies* catalog view. To achieve this task, it has to execute a *SqlCommand*. *SqlCommand* needs a connection. In the *GetEnvInfo* procedure's code you saw the usage of the *SqlContext* class; now you need an explicit *SqlConnection* object. You can get the context of the caller's connection by using a new connection string option, *"Context connection = true"*.

As in the *GetEnvInfo* procedure, you want to get the results in tabular format. Again you use the *SqlDataRecord* and *SqlMetaData* objects to shape the row returned. Remember that the *SqlPipe* object gives you the best performance to return the row to the caller.

Before you can read the metadata of an assembly, you have to load it. The rest is quite easy. The *GetTypes* method of a loaded assembly can be used to retrieve a collection of all types defined in the assembly. The code retrieves this collection in an array. Then it loops through the array, and for each type it uses the *GetMethods* method to retrieve all public methods in an array of the *MethodInfo* objects. This procedure retrieves type and method names only. The *Reflection* classes allow you to get other metadata information as well—for example, the names and types of input parameters. Here's the definition of the *GetAssemblyInfo* stored procedure using Visual Basic code:

```
' GetAssemblyInfo Procedure
' Returns assembly info, uses Reflection
<SqlProcedure()> _
Public Shared Sub GetAssemblyInfo(ByVal asmName As SqlString)
    ' Retrieve the clr name of the assembly
    Dim clrName As String = Nothing
    ' Get the context
    Using connection As New SqlConnection("Context connection = true")
        connection.Open()
        Using command As New SqlCommand
            ' Get the assembly and load it
            command.Connection = connection
            command.CommandText = _
              "SELECT clr_name FROM sys.assemblies WHERE name = @asmName"
            command.Parameters.Add("@asmName", SqlDbType.NVarChar)
            command.Parameters(0).Value = asmName
            clrName = CStr(command.ExecuteScalar())
            If (clrName = Nothing) Then
                Throw New ArgumentException("Invalid assembly name!")
            End If
            Dim myAsm As Assembly = Assembly.Load(clrName)
            ' Create a record - object representation of a row
            ' Include the metadata for the SQL table
            Dim record As New SqlDataRecord( _
                New SqlMetaData("Type", SqlDbType.NVarChar, 50), _
                New SqlMetaData("Name", SqlDbType.NVarChar, 256))
            ' Marks the beginning of the result set to be sent back
            ' to the client
            ' The record parameter is used to construct the metadata
            ' for the result set
            SqlContext.Pipe.SendResultsStart(record)
            ' Get all types in the assembly
            Dim typesArr() As Type = myAsm.GetTypes()
            For Each t As Type In typesArr
                ' Type in a SQL database should be a class or a structure
                If (t.IsClass = True) Then
                    record.SetSqlString(0, "Class")
                Else
                    record.SetSqlString(0, "Structure")
                End If
                record.SetSqlString(1, t.FullName)
                SqlContext.Pipe.SendResultsRow(record)
                ' Find all public static methods
                Dim miArr() As MethodInfo = t.GetMethods
                For Each mi As MethodInfo In miArr
                    If (mi.IsPublic And mi.IsStatic) Then
                        record.SetSqlString(0, "  Method")
                        record.SetSqlString(1, mi.Name)
```

```
                            SqlContext.Pipe.SendResultsRow(record)
                        End If
                    Next
                Next
                ' End of result set
                SqlContext.Pipe.SendResultsEnd()
            End Using
        End Using
    End Sub
```

Run the following code to register the C# version of the *GetAssemblyInfo* stored procedure in the CLRUtilities database:

```
IF OBJECT_ID('dbo.GetAssemblyInfo', 'PC') IS NOT NULL
  DROP PROC dbo.GetAssemblyInfo;
GO
CREATE PROCEDURE GetAssemblyInfo
  @asmName AS sysname
AS EXTERNAL NAME CLRUtilities.CLRUtilities.GetAssemblyInfo;
```

And in case you used Visual Basic to develop the stored procedure, use the following code to register it:

```
CREATE PROCEDURE GetAssemblyInfo
  @asmName AS sysname
AS EXTERNAL NAME
  CLRUtilities.[CLRUtilities.CLRUtilities].GetAssemblyInfo;
```

Run the following code to test the *GetAssemblyInfo* procedure, providing it with the CLRUtilities assembly name as input:

```
EXEC GetAssemblyInfo N'CLRUtilities';
```

You get the following output with the assembly name and the names of all methods (routines) defined within it:

```
Type       Name
---------- ----------------------
Class      CLRUtilities
  Method   RegexIsMatch
  Method   RegexReplace
  Method   FormatDatetime
  Method   ImpCast
  Method   ExpCast
  Method   SQLSigCLR
  Method   SplitCLR
  Method   ArrSplitFillRow
  Method   GetEnvInfo
  Method   GetAssemblyInfo
  Method   trg_GenericDMLAudit
  Method   SalesRunningSum
Structure  CLRUtilities+row_item
```

You should recognize most routine names except *trg_GenericDMLAudit* and *SalesRunningSum*, which will be covered later in the book.

When you're done, run the following code for cleanup:

```
USE CLRUtilities;
IF OBJECT_ID('dbo.GetEnvInfo', 'PC') IS NOT NULL
  DROP PROC dbo.GetEnvInfo;
IF OBJECT_ID('dbo.GetAssemblyInfo', 'PC') IS NOT NULL
  DROP PROC dbo.GetAssemblyInfo;
```

Conclusion

Stored procedures are one of the most powerful tools that SQL Server provides. Understanding them well and using them wisely will result in robust, secure databases that perform well. Stored procedures give you a security layer, encapsulation, reduction in network traffic, reuse of execution plans, and much more. SQL Server 2008 supports developing CLR routines, allowing you to enhance the functionality of your database.

Chapter 7

Temporary Tables and Table Variables

Itzik Ben-Gan

T-SQL programming often involves the need to materialize data temporarily. *Temporary tables* are just one solution; other ways for handling an independent physical or logical materialization of a set include table variables and table expressions such as views, inline user-defined functions (UDFs), derived tables, and common table expressions (CTEs).

You might need to physically persist interim states of your data for performance reasons, or just as a staging area. Examples of such scenarios include:

- Materializing aggregated data to some level of granularity (for example, employee and month), and issuing running, sliding, and other statistical reports against that data
- Materializing a result of a query for paging purposes
- Materializing result sets of interim queries, and querying the materialized data
- Materializing the result of a query with the GROUPING SETS, CUBE and ROLLUP options, and issuing queries against that data
- Walking through the output of a cursor and saving information you read or calculate per row for further manipulation
- Pivoting data from an Open Schema environment to a more traditional form, and issuing queries against the pivoted data
- Creating a result set that contains a hierarchy with additional attributes such as materialized paths or levels, and issuing reports against the result
- Holding data that needs to be scrubbed before it can be inserted

One of the benefits of materializing data in a temporary table is that it can be more compact than the base data, with preprocessed calculations, and you can index it when it might be inefficient or impractical to index all the base data. In terms of performance, you typically benefit from materializing the data when you need to access it multiple times, but in some cases, even when all you have is a single query against the data, you benefit.

You might also need to materialize interim sets logically in virtual temporary tables (table expressions) to develop solutions in a modular approach. I'll show examples in this chapter that address this need as well. Either way, there are many cases in which using temporary tables, table variables, or table expressions can be useful.

There's a lot of confusion around choosing the appropriate type of temporary object for a given task, and there are many myths regarding the differences between temporary tables and table variables. Furthermore, temporary tables and table variables are often misused because of lack of knowledge of efficient set-based programming.

In this chapter, I will try to provide you with a clear picture of how the different temporary object types behave, in which circumstances you should use each, and whether you should use them at all. At the end of the chapter, I'll provide a summary table (Table 7-1) that contrasts and compares the different types. This table covers the factors you should take into consideration before making your choice.

Temporary Tables

SQL Server supports two types of temporary tables: local and global. For the most part, I'll focus on local temporary tables because this is the type you would typically consider in the same situations as table variables and table expressions. I'll also describe global temporary tables, but these typically have different uses than local temporary tables.

Local Temporary Tables

I'll start with some fundamentals of local temporary tables before showing examples, and I'll do the same whenever discussing a new temporary object type. When referring to temporary tables in this section, assume that the discussion pertains to local ones.

You create and manipulate a temporary table just as you would a permanent one, for the most part. I'll point out the aspects of temporary tables that are different from permanent ones, or aspects that are often misunderstood.

tempdb

Temporary tables are created in tempdb, regardless of the database context of your session. They have physical representation in tempdb, although when they're small enough and Microsoft SQL Server has enough memory to spare, their pages reside in cache. SQL Server persists the temporary table's pages on disk when there is too little free memory. Furthermore, tempdb's recovery model is SIMPLE and cannot be changed. This means that bulk operations against temporary tables can benefit from minimal logging. Also, SQL Server supports a deferred drop feature in tempdb. When the application drops a large temporary table SQL Servers defers the drop activity to a background thread, so the application can continue working immediately.

Unlike user databases, tempdb is created from scratch as a copy of the model database every time you restart SQL Server, hence there's no need for a recovery process in tempdb. This fact leads to optimizations that you can benefit from when modifying data in tempdb regardless of the object type you are working with (temp table, table variable, or even a regular table). The transaction log doesn't need to be flushed to disk and therefore transactions in tempdb are committed faster. Also, certain types of modifications against objects in tempdb (mainly INSERT and UPDATE operations on heap and LOB data) can benefit from optimized logging: because you don't need to run a redo phase from the log (roll forward transactions that were committed after the last checkpoint) only the value before the change needs to be recorded in the log—not the value after the change. Later in the chapter I'll provide more details about working with tempdb.

One reason to use a temporary table is to take the load off of a user database when you need to persist temporary data. You can also enjoy the fact that tempdb is treated differently from user databases.

Tip My preferred method for checking whether an object already exists is to use the OBJECT_ID function. If the function returns NULL, the object doesn't exist. If you want to check whether a temporary table already exists, make sure you specify the tempdb database prefix; otherwise, SQL Server looks for it in the current database, doesn't find it, and always returns NULL. For example, to check whether #T1 exists, use *OBJECT_ID('tempdb..#T1')* and not *OBJECT_ID('#T1')*.

Also, SQL Server supports a second argument for OBJECT_ID, where you can specify the object type you're looking for (for example, 'U' for user table). The second argument's value must match the type column in *sys.objects*.

Scope and Visibility

Temporary table names are prefixed with a number symbol (#). A temporary table is owned by the creating session and visible only to it. However, SQL Server allows different sessions to create a temporary table with the same name. Internally, SQL Server adds underscores and a unique numeric suffix to the table name to distinguish between temporary tables with the same name across sessions. For example, suppose that you created a temporary table called #T1. If you query the view *sys.objects* in tempdb looking for a table with the name LIKE '#T1%', you will find a table with a name similar to the following (the suffix will vary):

#T1____00000000001E. Although this is the table's internal name, you refer to it in your code by the name you used when you created it—#T1.

Within the session, the temporary table is visible only to the creating level in the call stack and also inner levels, not to outer ones. For example, if you create a temp table in the session's outermost level, it's available anywhere within the session, across batches, and even in inner levels—for example, dynamic batch, stored procedure, and trigger. As long as you don't close the connection, you can access the temporary table. If it's created within a stored procedure, it's visible to the stored procedure and inner levels invoked by that procedure (for example, a nested procedure or a trigger). You can rely on the visibility behavior of temporary tables—for example, when you want to pass data between different levels in your session, or even just signal something to an inner level and that inner level doesn't support input parameters (for example, a trigger). However, in some cases, you can pass such information through the *context_info* feature, which is visible across the session. (See SET CONTEXT_INFO in SQL Server Books Online for details.)

When its creating level gets out of scope (terminates), a temporary table is automatically destroyed. If a temporary table was created in the outermost level, it is destroyed when the session is terminated. If it's created within a stored procedure, it is automatically dropped as soon as the stored procedure is finished.

Remember that a temporary table is not visible to levels outside of the creating one in the call stack. That's why, for example, you can't use a temporary table created in a dynamic batch in the calling batch. When the dynamic batch is out of scope, the temporary table is gone. Later in the chapter, I'll suggest alternatives to use when such a need occurs. The next part, regarding the scope, is a bit tricky. You can, in fact, create multiple temporary tables with the same name within the same session, as long as you create them in different levels—although doing so might lead to trouble. I'll elaborate on this point in the "Temporary Table Name Resolution" section later in the chapter.

The scope and visibility of a temporary table are very different than they are with both permanent tables and table variables and can be major factors in choosing one type of temporary object over another.

Transaction Context

A temporary table is an integral part of an outer transaction if it's manipulated in one (with DML or DDL). This fact has consequences for logging and locking. Logging has to support rollback operations only, not roll-forward ones. (Remember, there is no recovery process in tempdb.) As for locking, because the temporary table is visible only to the creating session, less locking is involved than with permanent tables, which can be accessed from multiple sessions.

Therefore, one of the factors you should consider when choosing a temporary object type is whether you want manipulation against it to be part of an outer transaction.

Statistics

The optimizer creates and maintains distribution statistics (column value histograms) for temporary tables and keeps track of their cardinality, much as it does for permanent ones. This capability is especially important when you index the temporary table. Distribution information is available to the optimizer when it needs to estimate selectivity, and you will get optimized plans that were generated based on this information. This is one of the main areas in which temporary tables differ from table variables in terms of performance.

Also, because statistics are maintained for temporary tables, queries against your temporary tables will be recompiled because of plan optimality reasons (recompilation threshold reached, statistics refreshed, and so on). The recompilation threshold is reached when a sufficient number of rows of a referenced table have changed since the last compilation. The recompilation threshold (RT) is based on the table type and the number of rows. For permanent tables, if $n \leq 500$, then $RT = 500$ (n = table's cardinality when a query plan is compiled). If $n > 500$, then $RT = 500 + 0.20 \times n$. For temporary tables, if $n < 6$, then $RT = 6$. If $6 \leq n \leq 500$, then $RT = 500$. *If* $n > 500$, then $RT = 500 + 0.20 \times n$. You realize that, for example, after inserting six rows into a temporary table, adding a seventh will trigger a recompile, whereas with permanent tables the first trigger will occur much later. If you want queries against temporary tables to use the same recompilation thresholds as against permanent ones, use the KEEP PLAN query hint.

The fact that the optimizer maintains distribution statistics for temporary tables and the aforementioned implications are the most crucial aspects of choosing a temporary object type. These factors are especially important when choosing between temporary tables and table variables, for which the optimizer doesn't create or maintain distribution statistics. Rowcount information is maintained for table variables (in *sys.partitions*) but this information is often inaccurate. Table variables themselves do not trigger recompiles because of plan optimality reasons, and recompiles are required to update the rowcount information. You can force a recompile for a query involving table variables using the RECOMPILE query hint.

You must ask yourself two main questions when considering which type of temporary object to use:

1. Does the optimizer need distribution statistics or accurate cardinality estimations to generate an efficient plan, and if so, what's the cost of using an inefficient plan when statistics are not available?
2. What's the cost of recompilations if you do use temporary tables?

In some cases the optimizer doesn't need statistics to figure out an optimal plan—for example, given a query requesting all rows from a table, a point query filtering a column on which a unique index is defined, a range query that utilizes a clustered or covering index, and so on. In such cases, regardless of the table's size, there's no benefit in having statistics because you will only suffer from the cost of recompilations. In such cases, consider using a table variable.

Also, if the table is tiny (say, a couple of pages), the alternatives are 1) using a table variable resulting in complete scans and few or no recompilations; or 2) using a temporary table resulting in index seeks and more recompilations. The advantage of seeks versus scans may be outweighed by the disadvantage of recompiles. That's another case for which you should consider using table variables.

On the other hand, if the optimizer does need statistics to generate an efficient plan and you're not dealing with tiny tables, the cost of using an inefficient plan might well be substantially higher than the cost of the recompilations involved. That's a case in which you should consider using temporary tables. In the "Table Variables" section, I'll provide examples related to these scenarios in which I'll also demonstrate execution plans.

Temporary Table Name Resolution

As I mentioned earlier, technically you're allowed to create multiple local temporary tables with the same name within the same session, as long as you create them in different levels. However, you should avoid doing this because of name-resolution considerations that might cause your code to break.

When a batch is resolved, the schema of a temporary table that is created within that batch is not available. So resolution of code that refers to the temporary table is deferred to run time. However, if a temporary table name you refer to already exists within the session (for example, it has been created by a higher level in the call stack), that table name will resolve to the existing temporary table. However, the code will always run against the innermost temporary table with the referenced name.

This resolution architecture can cause your code to break when you least expect it; this can happen when temporary tables with the same name exist in different levels with different schemas.

This part is very tricky and is probably best explained by using an example. Run the following code to create the stored procedures *proc1* and *proc2*:

```
SET NOCOUNT ON;
USE tempdb;

IF OBJECT_ID('dbo.proc1', 'P') IS NOT NULL DROP PROC dbo.proc1;
IF OBJECT_ID('dbo.proc2', 'P') IS NOT NULL DROP PROC dbo.proc2;
GO

CREATE PROC dbo.proc1
AS

CREATE TABLE #T1(col1 INT NOT NULL);
INSERT INTO #T1 VALUES(1);
SELECT * FROM #T1;

EXEC dbo.proc2;
GO

CREATE PROC dbo.proc2
AS

CREATE TABLE #T1(col1 INT NULL);
INSERT INTO #T1 VALUES(2);
SELECT * FROM #T1;
GO
```

proc1 creates a temporary table called #T1 with a single integer column, inserts a row with the value 1, returns #T1's contents, and invokes *proc2*, which also creates a temporary table called #T1 with a single integer column, inserts a row with the value 2, and returns #T1's contents. Both #T1 tables have the same schema. Now, invoke *proc1*:

```
EXEC dbo.proc1;
```

The output is what you probably expected:

```
col1
-----------
1

col1
-----------
2
```

Both procedures returned the contents of the #T1 table they created. Being oblivious to the resolution process I described earlier doesn't really affect you in this case. After all, you did get the expected result, and the code ran without errors. However, things change if you alter *proc2* in such a way that it creates #T1 with a different schema than in *proc1*:

```
ALTER PROC dbo.proc2
AS

CREATE TABLE #T1(col1 INT NULL, col2 INT NOT NULL);
INSERT INTO #T1 VALUES(2, 2);
SELECT * FROM #T1;
GO
```

Run *proc1* again:

```
EXEC dbo.proc1;
```

And notice the error you get in the output:

```
col1
-----------
1

Msg 213, Level 16, State 1, Procedure proc2, Line 5
Insert Error: Column name or number of supplied values does not match table definition.
```

Can you explain the error? Admittedly, the problem in the resolution process I described is very elusive, and you might not have realized it after the first read. Try to read the paragraph describing the resolution process again, and then see whether you can explain the error. Essentially, when *proc2* was invoked by *proc1*, a table called #T1 already existed. So even though *proc2*'s code creates a table called #T1 with two columns and inserts a row with two values, when the INSERT statement is resolved, *proc2*'s #T1 does not exist yet, but *proc1*'s does. Therefore, SQL Server reports a resolution error—you attempt to insert a row with two values to a table with one column (as if).

If you invoke *proc2 alone*, the code has no reason to fail because no other #T1 table exists in the session—and it doesn't fail:

```
EXEC dbo.proc2;
```

You get an output with the row loaded to *proc2*'s #T1:

```
col1        col2
----------- -----------
2           2
```

The execution plan for *proc2* now resides in cache. Ironically, if you now run *proc1* again, the code will complete without errors. *proc2* will not go through a resolution process again (neither will it go through parsing or optimization); rather, SQL Server simply reuses the plan from cache:

```
EXEC dbo.proc1;
```

And now you get the output you probably expected to begin with:

```
col1
-----------
1

col1        col2
----------- -----------
2           2
```

However, if *proc2*'s plan is removed from cache and you run *proc1*, your code will break:

```
EXEC sp_recompile 'dbo.proc2';
EXEC dbo.proc1;
```

This generates the following output:

```
Object 'dbo.proc2' was successfully marked for recompilation.
col1
-----------
1

Msg 213, Level 16, State 1, Procedure proc2, Line 5
Column name or number of supplied values does not match table definition.
```

In short, I hope that you realize it's wise to avoid naming temporary tables the same in different stored procedures/levels. A way to avoid such issues is to add a unique proc identifier to the names of temporary tables. For example, you could name the temporary table in *proc1* #T1_proc1, and in *proc2* name the temporary table #T1_proc2.

When you're done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.proc1', 'P') IS NOT NULL DROP PROC dbo.proc1;
IF OBJECT_ID('dbo.proc2', 'P') IS NOT NULL DROP PROC dbo.proc2;
```

Schema Changes to Temporary Tables in Dynamic Batches

Remember that a local temporary table created in a certain level is not visible to outer levels in the call stack. Occasionally, programmers look for ways around this limitation, especially when working with dynamic execution. That is, you want to construct the schema of the temporary table dynamically and populate it based on some user input, and then access it from an outer level. Frankly, insisting on using local temporary tables in such a scenario is very problematic. The solution involves ugly code, as is the nature of dynamic SQL in general, plus recompilations resulting from schema changes and data modifications. You should consider other alternatives to provide for the original need. Still, I want to show you a way around the limitations.

Here's an initial algorithm that attempts to provide a solution for this request:

1. In the outer level, create temporary table #T with a single dummy column.
2. Within a dynamic batch, perform the following tasks:
    a. Alter #T, adding the columns you need.
    b. Alter #T, dropping the dummy column.
    c. Populate #T.
3. Back in the outer level, access #T in a new batch.

The problem with this algorithm lies in the last item within the dynamic batch. References to #T will be resolved against the outer #T's schema. Remember that when the batch is resolved, #T's new schema is not available yet. The solution is to populate #T within another dynamic batch, in a level inner to the dynamic batch that alters #T's schema. You do this by performing the following tasks:

1. In the outer level, create temporary table #T with a single dummy column.
2. Within a dynamic batch, perform the following tasks:
    a. Alter #T, adding the columns you need.
    b. Alter #T, dropping the dummy column.
    c. Open another level of dynamic execution and within it populate #T.
3. Back in the outer level, access #T in a new batch.

Here's some sample code that implements this algorithm:

```
-- Assume @column_defs and @insert were constructed dynamically
-- with appropriate safeguards against SQL injection
DECLARE @column_defs AS VARCHAR(1000), @insert AS VARCHAR(1000);
SET @column_defs = 'col1 INT, col2 DECIMAL(10, 2)';
SET @insert = 'INSERT INTO #T VALUES(10, 20.30)';
```

```
-- In the outer level, create temp table #T with a single dummy column
CREATE TABLE #T(dummycol INT);

-- Within a dynamic batch:
--    Alter #T adding the columns you need
--    Alter #T dropping the dummy column
--    Open another level of dynamic execution
--      Populate #T
EXEC('
ALTER TABLE #T ADD ' + @column_defs + ';
ALTER TABLE #T DROP COLUMN dummycol;
EXEC(''' + @insert + ''')');
GO

-- Back in the outer level, access #T in a new batch
SELECT * FROM #T;

-- Cleanup
DROP TABLE #T;
```

This generates the following output:

```
col1        col2
----------- -----------
10          20.30
```

Caching of Temporary Objects

SQL Server 2008 supports the caching of temporary objects across repeated calls of routines. This feature is applicable to local temporary tables, table variables, and table-valued functions used within routines such as stored procedures, triggers, and user-defined functions. When the routine finishes, SQL Server keeps the catalog entry. If the object is smaller than 8 MB, SQL Server keeps one data page and one IAM page, and uses those instead of allocating new ones when the object is created again. If the object is larger than 8 MB, SQL Server uses deferred drop, and immediately returns control to the application. This feature results in reduction of contention against system catalog tables and allocation pages, and in faster creating and dropping of temporary objects.

I'll demonstrate caching of temporary objects (across repeated calls of routines) through an example. Run the following code to create a stored procedure called *TestCaching* that creates a temporary table called #T1 and populates it with a few rows:

```
SET NOCOUNT ON;
USE tempdb;

IF OBJECT_ID('dbo.TestCaching', 'P') IS NOT NULL
  DROP PROC dbo.TestCaching;
GO
CREATE PROC dbo.TestCaching
AS
```

```
CREATE TABLE #T1(n INT, filler CHAR(2000));

INSERT INTO #T1 VALUES
  (1, 'a'),
  (2, 'a'),
  (3, 'a');
GO
```

Run the following query to determine which entries representing temporary tables exist in the system catalog:

```
SELECT name FROM tempdb.sys.objects WHERE name LIKE '#%';
```

At this point there are no entries in the system catalog representing temporary tables; therefore, this query returns an empty set.

Execute the *TestCaching* procedure:

```
EXEC dbo.TestCaching;
```

The stored procedure terminated, but the temporary table was cached—or more specifically, SQL Server kept its entry in the system catalog, an IAM page, and a data page. Query *tempdb.sys.objects* again:

```
SELECT name FROM tempdb.sys.objects WHERE name LIKE '#%';
```

This time you get an entry back representing the temporary table that was cached. I got the following output (but of course you will get a different table name):

```
name
-----------
#2DE6D218
```

If the procedure's execution plan is recompiled or removed from cache, SQL Server removes the cached temporary objects that were created by the stored procedure from cache as well. SQL Server also removes cached temporary objects when tempdb has little free space.

Run the following code to mark the stored procedure for recompile, causing the associated cached temporary object to be removed from cache:

```
EXEC sp_recompile 'dbo.TestCaching';
```

Query *sys.objects*:

```
SELECT name FROM tempdb.sys.objects WHERE name LIKE '#%';
```

The query should return an empty result set. If not, try again in a few seconds, because the table is dropped in the background.

Note that in the following cases SQL Server will not cache temporary objects across procedure calls:

- When you issue a DDL statement against the temporary table after it was created "e.g., CREATE INDEX".
- When you define a named constraint.
- When you create the temporary object in a dynamic batch within the routine.
- When you create the temporary object in an ad-hoc batch (not within a routine).

I'll first demonstrate the effect of applying DDL changes post-creation of the temporary table. Run the following code to alter the procedure *TestCaching*, adding an index to the temporary table after it was created:

```
ALTER PROC dbo.TestCaching
AS

CREATE TABLE #T1(n INT, filler CHAR(2000));
CREATE UNIQUE INDEX idx1 ON #T1(n);

INSERT INTO #T1 VALUES
  (1, 'a'),
  (2, 'a'),
  (3, 'a');
GO
```

Next, run the procedure and query *sys.objects*:

```
EXEC dbo.TestCaching;
SELECT name FROM tempdb.sys.objects WHERE name LIKE '#%';
```

This returns an empty result set, indicating that the temporary table wasn't cached.

As a workaround, you can include an unnamed UNIQUE or PRIMARY KEY constraint as part of the temporary table definition. The constraint implicitly creates a unique index on the constraint keys. Run the following code to test this approach:

```
ALTER PROC dbo.TestCaching
AS

CREATE TABLE #T1(n INT, filler CHAR(2000), UNIQUE(n));

INSERT INTO #T1 VALUES
  (1, 'a'),
  (2, 'a'),
  (3, 'a');
GO

EXEC dbo.TestCaching;

SELECT name FROM tempdb.sys.objects WHERE name LIKE '#%';
```

This time the query against *sys.objects* should report one temporary table. I got the following output:

```
name
-----------
#3A4CA8FD
```

Note that you can create composite indexes implicitly without sacrificing caching by including a composite UNIQUE or PRIMARY KEY constraint in your table definition, as in UNIQUE(col1, col2, col3).

As for named constraints, you might find this restriction odd, but naming a constraint prevents SQL Server from caching your temporary objects. You just saw in the last example that when the UNIQUE constraint was not named, SQL Server cached the temporary table. Now try the same example, but this time name the constraint:

```
ALTER PROC dbo.TestCaching
AS

CREATE TABLE #T1(n INT, filler CHAR(2000), CONSTRAINT UNQ_#T1_n UNIQUE(n));

INSERT INTO #T1 VALUES
  (1, 'a'),
  (2, 'a'),
  (3, 'a');
GO

EXEC dbo.TestCaching;

SELECT name FROM tempdb.sys.objects WHERE name LIKE '#%';
```

This time the temporary object wasn't cached (again, it may take a few seconds for the temporary object that was cached previously to be removed from cache). So even though naming constraints is in general a good practice, bear in mind that if you want to benefit from caching of temporary objects, you shouldn't name them.

Global Temporary Tables

Global temporary tables differ from local ones mainly in their scope and visibility. They are accessible by all sessions, with no security limitations whatsoever. Any session can even drop the table. So when you design your application, you should factor in security and consider whether you really want temporary tables or just permanent ones. You create global temporary tables by prefixing their names with two number signs (##), and like local temporary tables, they are created in tempdb. However, because global temporary tables are accessible to all sessions, you cannot create multiple ones with the same name; neither in the same session nor across sessions. So typical scenarios for using global temporary tables are when you want to share temporary data among sessions and don't care about security.

Unlike local temporary tables, global ones persist until the creating session—not the creating level—terminates. For example, if you create such a table in a stored procedure and the stored procedure goes out of scope, the table is not destroyed. SQL Server will automatically attempt to drop the table when the creating session terminates, all statements issued against it from other sessions finish, and any locks they hold are released.

I'll walk you through a simple example to demonstrate the accessibility and termination of a global temporary table. Open two connections to SQL Server (call them Connection 1 and Connection 2). In Connection 1, create and populate the table ##T1:

```
CREATE TABLE ##T1(col1 INT);
INSERT INTO ##T1 VALUES(1);
```

In Connection 2, open a transaction and modify the table:

```
BEGIN TRAN
  UPDATE ##T1 SET col1 = col1 + 1;
```

Then close Connection 1. If not for the open transaction that still holds locks against the table, SQL Server would have dropped the table at this point. However, because Connection 2 still holds locks against the table, it's not dropped yet. Next, in Connection 2, query the table and commit the transaction:

```
  SELECT * FROM ##T1;
COMMIT
```

At this point, SQL Server drops the table because no active statements are accessing it, and no locks are held against it. If you try to query it again from any session, you will get an error saying that the table doesn't exist:

```
SELECT * FROM ##T1;
```

In one special case you might want to have a global temporary table available but not owned by any session. In this case, it will always exist, regardless of which sessions are open or closed, and eliminated only if someone explicitly drops it. To achieve this, you create the table within a procedure, and mark the stored procedure with the *startup* procedure option. SQL Server invokes a startup procedure every time it starts. Furthermore, SQL Server always maintains a reference counter greater than zero for a global temporary table created within a startup procedure. This ensures that SQL Server will not attempt to drop it automatically.

Here's some sample code that creates a startup procedure called *CreateGlobals*, which in turn creates a global temporary table called ##Globals.

```
USE master;
IF OBJECT_ID('dbo.CreateGlobals', 'P') IS NOT NULL DROP PROC dbo.CreateGlobals
GO
CREATE PROC dbo.CreateGlobals
AS
```

```
CREATE TABLE ##Globals
(
  varname sysname NOT NULL PRIMARY KEY,
  val     SQL_VARIANT NULL
);
GO

EXEC dbo.sp_procoption 'dbo.CreateGlobals', 'startup', 'true';
```

After restarting SQL Server, the global temporary table will be created automatically and persist until someone explicitly drops it. To test the procedure, restart SQL Server and then run the following code:

```
SET NOCOUNT ON;
INSERT INTO ##Globals VALUES('var1', CAST('abc' AS VARCHAR(10)));
SELECT * FROM ##Globals;
```

You probably guessed already that ##Globals is a shared global temporary table where you can logically maintain cross-session global variables. This can be useful, for example, when you need to maintain temporary counters or other "variables" that are globally accessible by all sessions. The preceding code creates a new global variable called *var1*, initializes it with the character string *'abc'*, and queries the table. Here's the output of this code:

```
varname     val
----------- -----------
var1        abc
```

When you're done, run the following code for cleanup:

```
USE master;
DROP PROC dbo.CreateGlobals;
DROP TABLE ##Globals;
```

Table Variables

Table variables are probably among the least understood T-SQL elements. Many myths and misconceptions surround them, and these are embraced even by experienced T-SQL programmers. One widespread myth is that table variables are memory-resident only, without physical representation. Another myth is that table variables are always preferable to temporary tables. In this section, I'll dispel these myths and explain the scenarios in which table variables are preferable to temporary tables as well as scenarios in which they aren't preferable. I'll do so by first going through the fundamentals of table variables, just as I did with temporary tables, and follow with tangible examples.

You create a table variable using a DECLARE statement, followed by the variable name and the table definition. You then refer to it as you do with permanent tables. Here's a very basic example:

```
DECLARE @T1 TABLE(col1 INT);
INSERT @T1 VALUES(1);
SELECT * FROM @T1;
```

Note that the table-valued parameters that were added in SQL Server 2008 are implemented internally like table variables. So the performance discussions in this section regarding table variables apply to table-valued parameters as well. Table-valued parameters were discussed earlier in the book in Chapter 3, "Stored Procedures."

Limitations

Many limitations apply to table variables but not to temporary tables. In this section, I'll describe some of them, whereas others will be described in dedicated sections.

- You cannot create explicit indexes on table variables, only PRIMARY KEY and UNIQUE constraints, which create unique indexes underneath the covers. You cannot create non-unique indexes. If you need an index on a non-unique column, you must add attributes that make the combination unique and create a PRIMARY KEY or UNIQUE constraint on the combination.
- You cannot alter the definition of a table variable once it is declared. This means that everything you need in the table definition must be included in the original DECLARE statement. This fact is limiting on one hand, but it also results in fewer recompilations. Remember that one of the triggers of recompilations is DDL changes.
- You cannot issue SELECT INTO against a table variable, rather you have to use INSERT SELECT instead. Prior to SQL Server 2008 this limitation put table variables at a disadvantage compared to temporary tables because SELECT INTO could be done as a minimally logged operation, though INSERT SELECT couldn't. SQL Server 2008 introduces improvements in minimally logged operations, including the ability to process INSERT SELECT with minimal logging. I'll demonstrate this capability later in the chapter.
- You cannot qualify a column name with a nondelimited table variable name (as in @T1.col1). This is especially an issue when referring to a table variable's column in correlated subqueries with column name ambiguity. To circumvent this limitation, you have to delimit the table variable name (as in [@T1].col1, or "@T1".col1).
- In queries that modify table variables, parallel plans will not be used. Queries that only read from table variables can be parallelized.

tempdb

To dispel what probably is the most widespread myth involving table variables, let me state that they do have physical representation in tempdb, very similar to temporary tables.

As proof, run the following code that shows which temporary tables currently exist in tempdb by querying metadata info, creating a table variable, and querying metadata info again:

```
SELECT TABLE_NAME
FROM tempdb.INFORMATION_SCHEMA.TABLES
WHERE TABLE_NAME LIKE '#%';
GO
DECLARE @T TABLE(col1 INT);

INSERT INTO @T VALUES(1);

SELECT TABLE_NAME
FROM tempdb.INFORMATION_SCHEMA.TABLES
WHERE TABLE_NAME LIKE '#%';
```

When I ran this code, the first batch returned no output, whereas the second returned #0CBAE877, which is the name of the temporary table in tempdb that represents the table variable *@T*. Of course, you will probably get a different name when you run this code. But the point is to show that a hidden temporary table is created behind the scenes. Just like temporary tables, a table variable's pages reside in cache when the table is small enough and when SQL Server has enough memory to spare. So the discussion about aspects of working with temporary tables with regard to tempdb applies to table variables as well.

Scope and Visibility

The scope of a table variable is well defined. It is defined as the current level, and within it the current batch only, just as with any other variable. That is, a table variable is not accessible to inner levels, and not even to other batches within the same level. In short, you can use it only within the same batch it was created. This scope is much more limited than that of a local temporary table and is typically an important factor in choosing a temporary object type.

Transaction Context

Unlike a temporary table, a table variable is not part of an outer transaction; rather, its transaction scope is limited to the statement level to support statement rollback capabilities only. If you modify a table variable and the modification statement is aborted, the changes of that particular statement will be undone. However, if the statement is part of an outer transaction that is rolled back, changes against the table variable that finished will not be undone. Table variables are unique in this respect.

You can rely on this behavior to your advantage. For example, suppose that you need to write an audit trigger that audits changes against some table. If some logical condition is met, you want to roll back the change; however, you still want to audit the attempted change. If you copy data from inserted/deleted to your audit tables, a rollback in the trigger also undoes the audit writes. If you first roll back the change and then try to audit it, deleted and inserted are empty.

To solve the problem, you first copy data from inserted/deleted to table variables, issue a rollback, and then in a new transaction within the trigger, copy the data from the table variables to your audit tables. This is the simplest way around the problem.

The unique transaction context of table variables has performance advantages over temporary tables because less logging and locking are involved.

Statistics

As I mentioned earlier, SQL Server doesn't create distribution statistics or maintain accurate cardinality information for table variables as it does for temporary tables. This is one of the main factors you should consider when choosing a type of temporary object for a given task. The downside is that you might get inefficient plans when the optimizer needs to consult histograms to determine selectivity. This is especially a problem with big tables, where you might end up with excessive I/O. The upside is that table variables, for the very same reason, involve much fewer recompilations. Before making your choice, you need to figure out which is more expensive in the particular task for which you're designating the temporary object.

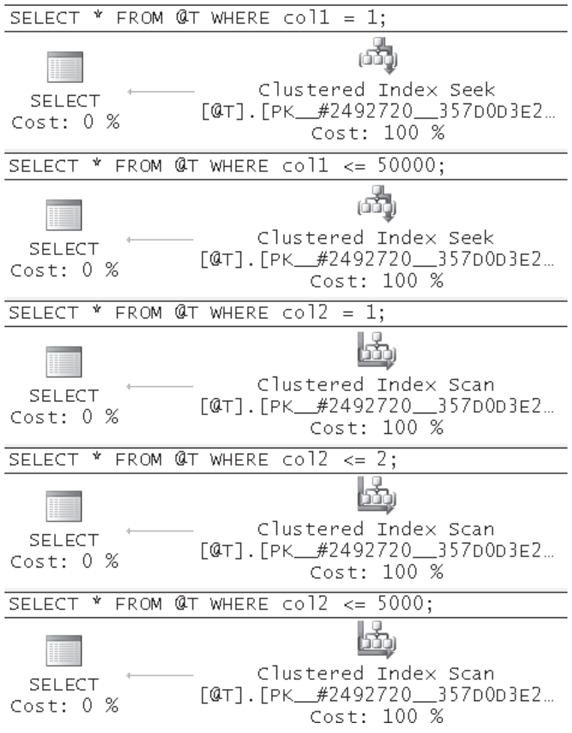
To explain the statistics aspect of table variables in a more tangible way, I'll show you some queries, their execution plans, and their I/O costs.

Examine the following code, and request an estimated execution plan for it from SQL Server Management Studio (SSMS):

```
DECLARE @T TABLE
(
  col1 INT NOT NULL PRIMARY KEY,
  col2 INT NOT NULL,
  filler CHAR(200) NOT NULL DEFAULT('a'),
  UNIQUE(col2, col1)
);

INSERT INTO @T(col1, col2)
  SELECT n, (n - 1) % 10000 + 1 FROM dbo.Nums
  WHERE n <= 100000;

SELECT * FROM @T WHERE col1 = 1;
SELECT * FROM @T WHERE col1 <= 50000;

SELECT * FROM @T WHERE col2 = 1;
SELECT * FROM @T WHERE col2 <= 2;
SELECT * FROM @T WHERE col2 <= 5000;
```

You can find the code to create and populate the Nums table in Chapter 2, "User-Defined Functions."

The estimated execution plans generated for these queries are shown in Figure 7-1.



```
SELECT * FROM @T WHERE col1 = 1;
SELECT
Cost: 0 %        Clustered Index Seek
                 [@T].[PK__#2492720__357D0D3E2…
                 Cost: 100 %
SELECT * FROM @T WHERE col1 <= 50000;
SELECT
Cost: 0 %        Clustered Index Seek
                 [@T].[PK__#2492720__357D0D3E2…
                 Cost: 100 %
SELECT * FROM @T WHERE col2 = 1;
SELECT
Cost: 0 %        Clustered Index Scan
                 [@T].[PK__#2492720__357D0D3E2…
                 Cost: 100 %
SELECT * FROM @T WHERE col2 <= 2;
SELECT
Cost: 0 %        Clustered Index Scan
                 [@T].[PK__#2492720__357D0D3E2…
                 Cost: 100 %
SELECT * FROM @T WHERE col2 <= 5000;
SELECT
Cost: 0 %        Clustered Index Scan
                 [@T].[PK__#2492720__357D0D3E2…
                 Cost: 100 %
```

FIGURE 7-1 Estimated execution plans for queries against a table variable

The code creates a table variable called *@T* with two columns. The values in *col1* are unique, and each value in *col2* appears 10 times. The code creates two unique indexes underneath the covers: one on *col1*, and one on *(col2, col1)*.

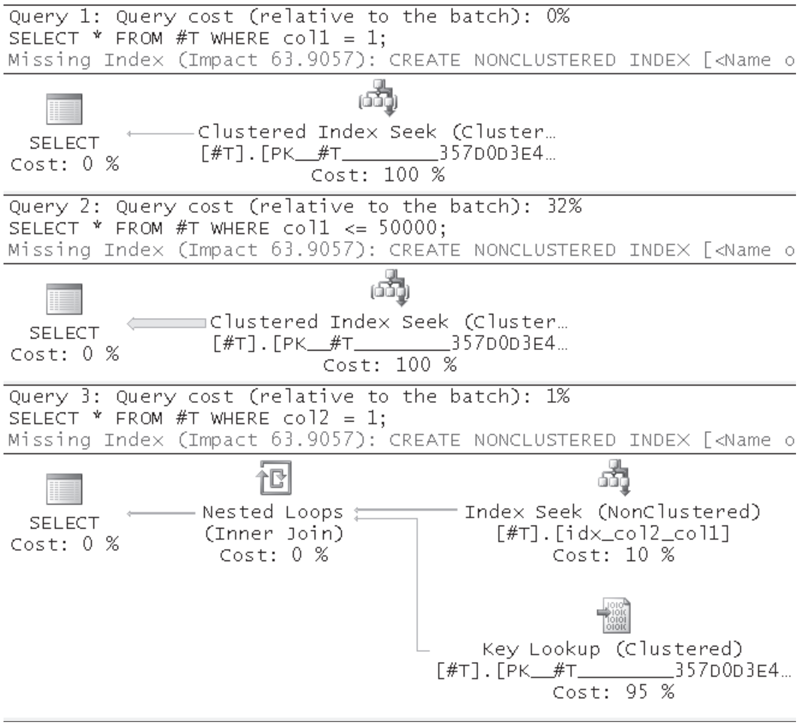
The first important thing to notice in the estimated plans is the number of rows the optimizer estimates to be returned from each operator—one in all five cases, even when looking for a non-unique value or ranges. You realize that unless you filter a unique column, the optimizer simply cannot estimate the selectivity of queries for lack of statistics. So it assumes one row. This hard-coded assumption is based on the fact that SQL Server assumes that you use table variables only with small sets of data.

As for the efficiency of the plans, the first two queries get a good plan (seek, followed by a partial scan in the second query). But that's because you have a clustered index on the filtered column, and the optimizer doesn't need statistics to figure out what the optimal plan is in this case. However, with the third and fourth queries you get a table scan (an unordered clustered index scan) even though both queries are very selective and would benefit from using the index on *(col2, col1)*, followed by a small number of lookups. The fifth query would benefit from a table scan because it has low selectivity. Fortunately, it got an adequate plan, but that's by chance. To analyze I/O costs, run the code after turning on the SET STATISTICS IO option. The amount of I/O involved with each of the last three queries is 2,713 reads, which is equivalent to the number of pages consumed by the table.

Next, go through the same analysis process with the following code, which uses a temporary table instead of a table variable:

```
SELECT n AS col1, (n - 1) % 10000 + 1 AS col2,
  CAST('a' AS CHAR(200)) AS filler
INTO #T
FROM dbo.Nums
WHERE n <= 100000;

ALTER TABLE #T ADD PRIMARY KEY(col1);
CREATE UNIQUE INDEX idx_col2_col1 ON #T(col2, col1);
GO

SELECT * FROM #T WHERE col1 = 1;
SELECT * FROM #T WHERE col1 <= 50000;

SELECT * FROM #T WHERE col2 = 1;
SELECT * FROM #T WHERE col2 <= 2;
SELECT * FROM #T WHERE col2 <= 5000;
```

The estimated execution plans generated for these queries are shown in Figures 7-2 and 7-3.



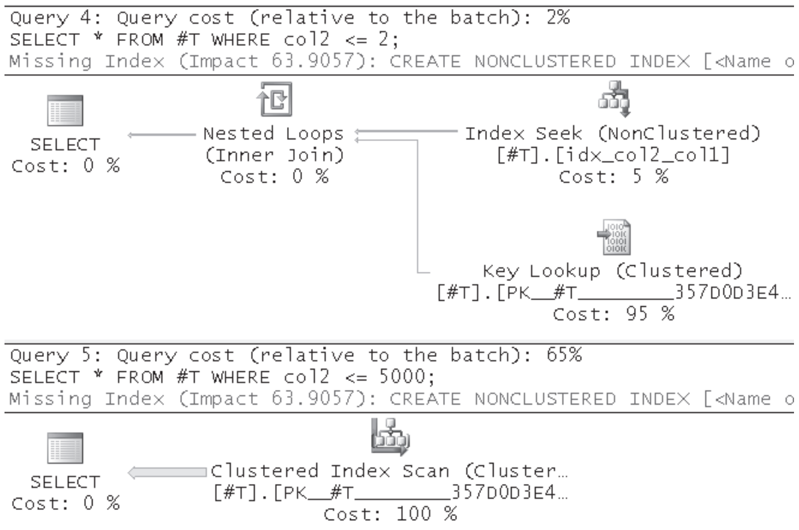
```
Query 1: Query cost (relative to the batch): 0%
SELECT * FROM #T WHERE col1 = 1;
Missing Index (Impact 63.9057): CREATE NONCLUSTERED INDEX [<Name o
```

```
Query 2: Query cost (relative to the batch): 32%
SELECT * FROM #T WHERE col1 <= 50000;
Missing Index (Impact 63.9057): CREATE NONCLUSTERED INDEX [<Name o
```

```
Query 3: Query cost (relative to the batch): 1%
SELECT * FROM #T WHERE col2 = 1;
Missing Index (Impact 63.9057): CREATE NONCLUSTERED INDEX [<Name o
```

SELECT
Cost: 0 %
Clustered Index Seek (Cluster…
[#T].[PK__#T________357D0D3E4…
Cost: 100 %
SELECT
Cost: 0 %
Clustered Index Seek (Cluster…
[#T].[PK__#T________357D0D3E4…
Cost: 100 %
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[#T].[idx_col2_col1]
Cost: 10 %
Key Lookup (Clustered)
[#T].[PK__#T________357D0D3E4…
Cost: 95 %

FIGURE 7-2 Estimated execution plans for queries 1, 2, and 3 against a temporary table


```
Query 4: Query cost (relative to the batch): 2%
SELECT * FROM #T WHERE col2 <= 2;
Missing Index (Impact 63.9057): CREATE NONCLUSTERED INDEX [<Name o
```

```
Query 5: Query cost (relative to the batch): 65%
SELECT * FROM #T WHERE col2 <= 5000;
Missing Index (Impact 63.9057): CREATE NONCLUSTERED INDEX [<Name o
```
SELECT
Cost: 0 %
Clustered Index Scan (Cluster…
[#T].[PK__#T__________357D0D3E4…
Cost: 100 %

FIGURE 7-3 Estimated execution plans for queries 4 and 5 against a temporary table

As an aside, in case you're curious about the Missing Index messages, SSMS 2008 reports this information in the graphical execution plan. Both SQL Server 2005 and SQL Server 2008 may enter a phase in optimization where they report missing index info. In both versions this information is available in the XML form of the execution plan. The new feature in SSMS 2008 is that it exposes this info graphically with the green-colored messages, whereas SSMS 2005 didn't.

Now that statistics are available, the optimizer can make educated estimations. You can see that the estimated number of rows returned from each operator is more reasonable. You can also see that high-selectivity queries 3 and 4 use the index on *(col2, col1)*, and the low-selectivity query 5 does a table scan, as it should.

STATISTICS IO reports dramatically reduced I/O costs for queries 3 and 4. These are 32 and 62 reads, respectively, against the temporary table versus 2,713 for each of these queries against the table variable.

When you're done, drop #T for cleanup:

```
DROP TABLE #T;
```

Minimally Logged Inserts

As mentioned earlier, you can use SELECT INTO with temporary tables but not with table variables. With table variables you have to use INSERT SELECT instead. Prior to SQL Server 2008, INSERT SELECT involved more logging than SELECT INTO. This was true even with the reduced logging that happens with inserts against objects in tempdb. SQL Server 2008 adds the INSERT SELECT statement to the list of insertion methods that can be performed in a minimally logged mode, just like SELECT INTO.

I'll demonstrate this capability through an example. I'll insert data into the temporary object using SELECT INTO and INSERT SELECT in both SQL Server 2005 and SQL Server 2008. To figure out the amount of logging involved with the operation, I'll query the undocumented *fn_dblog* function before and after the operation, and calculate the differences in terms of number of log records, and total record lengths, like so:

```
CHECKPOINT;
GO

DECLARE @numrecords AS INT, @size AS BIGINT;

SELECT
  @numrecords = COUNT(*),
  @size       = COALESCE(SUM([Log Record Length]), 0)
FROM fn_dblog(NULL, NULL) AS D;

-- <operation>

SELECT
  COUNT(*) - @numrecords AS numrecords,
  CAST((COALESCE(SUM([Log Record Length]), 0) - @size)
    / 1024. / 1024. AS NUMERIC(12, 2)) AS size_mb
FROM fn_dblog(NULL, NULL) AS D;
```

The first test is with the SELECT INTO statement that is processed with minimal logging in both SQL Server 2005 and SQL Server 2008, provided that the recovery model of the database is not set to FULL. As a reminder, tempdb's recovery model is SIMPLE and cannot be changed. Here's the code I used for this test:

```
USE tempdb;
CHECKPOINT;
GO

DECLARE @numrecords AS INT, @size AS BIGINT;

SELECT
  @numrecords = COUNT(*),
  @size       = COALESCE(SUM([Log Record Length]), 0)
FROM fn_dblog(NULL, NULL) AS D;

SELECT n, CAST('a' AS CHAR(2000)) AS filler
INTO #TestLogging
FROM dbo.Nums
WHERE n <= 100000;

SELECT
  COUNT(*) - @numrecords AS numrecords,
  CAST((COALESCE(SUM([Log Record Length]), 0) - @size)
    / 1024. / 1024. AS NUMERIC(12, 2)) AS size_mb
FROM fn_dblog(NULL, NULL) AS D;
GO

DROP TABLE #TestLogging;
```

As you can see, the operation is a SELECT INTO statement populating the temporary table #TestLogging with 100,000 rows by querying the Nums table. The output I got in SQL Server 2005 and SQL Server 2008 was similar:

```
numrecords  size_mb
----------- --------
9560        0.63
```

The number of log records is far lower than the number of rows inserted because only changes in allocation bitmaps (GAM, SGAM, PFS, IAM) were recorded in the log. Also, the total size recorded in the log is very small.

Next, I used the following code to test an INSERT SELECT against a table variable populating it with the same sample data used in the SELECT INTO test:

```
USE tempdb;
CHECKPOINT;
GO

DECLARE @numrecords AS INT, @size AS BIGINT;

SELECT
  @numrecords = COUNT(*),
  @size       = COALESCE(SUM([Log Record Length]), 0)
FROM fn_dblog(NULL, NULL) AS D;

DECLARE @TestLogging AS TABLE(n INT, filler CHAR(2000));

INSERT INTO @TestLogging(n, filler)
  SELECT n, CAST('a' AS CHAR(2000))
  FROM dbo.Nums
  WHERE n <= 100000;

SELECT
  COUNT(*) - @numrecords AS numrecords,
  CAST((COALESCE(SUM([Log Record Length]), 0) - @size)
    / 1024. / 1024. AS NUMERIC(12, 2)) AS size_mb
FROM fn_dblog(NULL, NULL) AS D;
GO
```

Here's the output I got in SQL Server 2005, indicating more logging activity than the corresponding SELECT INTO method:

```
numrecords  size_mb
----------- --------
184394      12.92
```

In SQL Server 2008 the output of the INSERT SELECT method was similar to the output I got for the corresponding SELECT INTO test, indicating minimal logging in both cases:

```
numrecords  size_mb
----------- --------
9539        0.63
```

This improvement in SQL Server 2008 means that temporary tables don't have an advantage over table variables in terms of amount of logging of SELECT INTO versus INSERT SELECT.

tempdb Considerations

Remember that temporary tables and table variables are physically stored in tempdb. SQL Server also stores data in tempdb for many implicit activities that take place behind the scenes. Examples for such activities include: spooling data as part of an execution plan of a query, sorting, hashing, and maintaining row versions. You realize that tempdb can become a bottleneck, and you should give it focused tuning attention so that it will accommodate the workload against your server.

Here are some important points you should consider when tuning tempdb:

- In systems where tempdb is heavily used (explicitly or implicitly), consider placing tempdb on its own disk array, and not on the same drives where other databases are located. Also, stripe the data portion to multiple drives to increase I/O throughput. The more spindles, the better. Ideally, use RAID 10 for the data portion and RAID 1 for the log.
- Every time you restart SQL Server, tempdb is re-created, and its size reverts to the effective defined size. If you made no changes to the original size configuration after installing SQL Server, tempdb's size will default to 8 MB and its growth increment will default to 10 percent. In most production environments, these values might not be practical. Whenever a process needs to store data in tempdb and tempdb is full, SQL Server will initiate an autogrow operation. The process will have to wait for the space to be allocated. Also, when the database is small, 10 percent is a very small unit. The small fragments will most probably be allocated in different places on disk, resulting in a high level of file-system fragmentation. And if that's not enough, remember that every time SQL Server restarts, tempdb's size will revert to its defined size (8 MB). This means that the whole process will start again, where tempdb will keep on autogrowing until it reaches a size appropriate to your environment's workload. Until it reaches that point, processes will suffer as they wait while tempdb autogrows.
- You can figure out the appropriate size for tempdb by observing its actual size after a period of activity without restarts. You then alter the database and change the SIZE parameter of tempdb's files so that tempdb's size will be appropriate. Whenever SQL Server is restarted, tempdb will just start out at the defined size. If you do this, there won't be a need for autogrowth until tempdb gets full, which should occur only with irregular and excessive tempdb activity.
- Remember that logically tempdb is re-created whenever SQL Server restarts. Like any other new database, tempdb is created as a copy of the model database. This means that if you create permanent objects in tempdb (permanent tables, user-defined types,

database users, and so on), they're erased in the next restart. If you need objects to exist in tempdb after restarts, you have two options. One is to create them in model. They will appear in tempdb after a restart. However, this option will also affect new user databases you create. Another option is to encapsulate code that creates all objects in a startup procedure. (See information on startup procedures earlier in the chapter in the "Global Temporary Tables" section.) Remember that a startup procedure is invoked whenever SQL Server is restarted. Essentially the objects will be re-created every time upon restart, but this will be invisible to users.

- With regard to temporary tables, obviously dealing with very large volumes of data can cause performance problems. However, you might face performance problems with tempdb even when working with small temporary tables. When many concurrent sessions create temporary tables, SQL Server might experience latch contention on allocation bitmaps when it tries to allocate pages. In the last couple of versions of SQL Server this problem was reduced substantially because of improvements in the engine—caching of temporary objects across routine calls—and improvements in the proportional fill algorithm SQL Server uses. Still, the problem may occur. The recommended practices to mitigate the problem are to use multiple data files for tempdb (as a general rule of thumb, one file per each CPU core), and to meet the requirements described earlier that would allow caching of temporary objects across routine calls.

More Info You can find more details about tempdb in papers found at the following URLs: *http://technet.microsoft.com/en-us/library/cc966545.aspx* and *http://technet.microsoft.com/en-us/library/cc966425.aspx*. Even though the papers were originally written for SQL Server 2005, most of the content describing SQL Server 2005 behavior is applicable for SQL Server 2008 as well.

Table Expressions

In this chapter's opening paragraphs, I mentioned that there might be cases in which you need "logical" temporary tables—that is, only virtual materialization of interim sets, as opposed to physical materialization in temporary tables and table variables. Table expressions give you this capability. These include derived tables, CTEs, views, and inline table-valued UDFs. Here I'll point out the scenarios in which these are preferable to other temporary objects and provide an example.

You should use table expressions in cases where you need a temporary object mainly for simplification—for example, when developing a solution in a modular approach, a step at a time. Also, use table expressions when you need to access the temporary object only once or a very small number of times and you don't need to index interim result sets. SQL Server doesn't physically materialize a table expression. The optimizer merges the outer query with the inner one, and it generates one plan for the query accessing the underlying tables directly. So I'm mainly talking about simplification, and I show such examples throughout the book. But even beyond simplification, in some cases you will be able to improve performance of solutions by using table expressions. There might be cases where the optimizer will generate a better plan for your query compared to alternative queries.

In terms of scope and visibility, derived tables and CTEs are available only to the current statement, whereas views and inline UDFs are available globally to users that have permissions to access them.

As an example of using a table expression to solve a problem, suppose you want to return from the Sales.Orders table in the InsideTSQL2008 database, the row with the highest *orderid* for each employee. Here's a solution that uses a CTE:

```
USE InsideTSQL2008;

WITH EmpMax AS
(
  SELECT empid, MAX(orderid) AS maxoid
  FROM Sales.Orders
  GROUP BY empid
)
SELECT O.orderid, O.empid, O.custid, O.orderdate
FROM Sales.Orders AS O
  JOIN EmpMax AS EM
    ON O.orderid = EM.maxoid;
```

This generates the following output:

```
orderid     empid       custid      orderdate
----------- ----------- ----------- -----------------------
11077       1           65          2008-05-06 00:00:00.000
11073       2           58          2008-05-05 00:00:00.000
11063       3           37          2008-04-30 00:00:00.000
11076       4           9           2008-05-06 00:00:00.000
11043       5           74          2008-04-22 00:00:00.000
11045       6           10          2008-04-23 00:00:00.000
11074       7           73          2008-05-06 00:00:00.000
11075       8           68          2008-05-06 00:00:00.000
11058       9           6           2008-04-29 00:00:00.000
```

Comparison Summary

Table 7-1 contains a summary of the functionality and behavior of the different object types. Note that I don't include global temporary tables because typically you use those for different purposes than the other types of temporary objects. You might find this table handy as a reference when you need to choose the appropriate temporary object type for a given task.

TABLE 7-1 **Comparison Summary**

Functionality/Object Type	Local Temp Table	Table Variable	Table Expression
Scope/Visibility	Current and inner levels	Local Batch	Derived Table/CTE: Current statement View/Inline UDF: Global
Physical representation in tempdb	Yes	Yes	No
Part of outer transaction/ affected by outer transaction rollback	Yes	No	N/A
Logging and locking	To support transaction rollback	To support statement rollback	N/A
Statistics/recompilations/ efficient plans	Yes	No	N/A
Table size	Any	Typically recommended for small tables	Any

Summary Exercises

This section will introduce three scenarios in which you need to work with temporary objects. Based on the knowledge you've acquired in this chapter, you need to implement a solution with the appropriate temporary object type.

The scenarios involve querying Customers and Orders tables. To test the logical correctness of your solutions, use the Sales.Customers and Sales.Orders tables in the InsideTSQL2008 sample database. To test the performance of your solutions, use the tables that you create and populate in tempdb by running the code in Listing 7-1.

LISTING 7-1 Code that creates large tables for summary exercises

```
SET NOCOUNT ON;
USE tempdb;

IF SCHEMA_ID('Sales') IS NULL EXEC('CREATE SCHEMA Sales');
IF OBJECT_ID('Sales.Customers', 'U') IS NOT NULL DROP TABLE Sales.Customers;
IF OBJECT_ID('Sales.Orders', 'U') IS NOT NULL DROP TABLE Sales.Orders;
GO

SELECT n AS custid
INTO Sales.Customers
FROM dbo.Nums
WHERE n <= 10000;
```

```
ALTER TABLE Sales.Customers ADD PRIMARY KEY(custid);

SELECT n AS orderid,
  DATEADD(day, ABS(CHECKSUM(NEWID())) % (4*365), '20060101') AS orderdate,
  1 + ABS(CHECKSUM(NEWID())) % 10000 AS custid,
  1 + ABS(CHECKSUM(NEWID())) % 40    AS empid,
  CAST('a' AS CHAR(200)) AS filler
INTO Sales.Orders
FROM dbo.Nums
WHERE n <= 1000000;

ALTER TABLE Sales.Orders ADD PRIMARY KEY(orderid);
CREATE INDEX idx_cid_eid ON Sales.Orders(custid, empid);
```

Comparing Periods

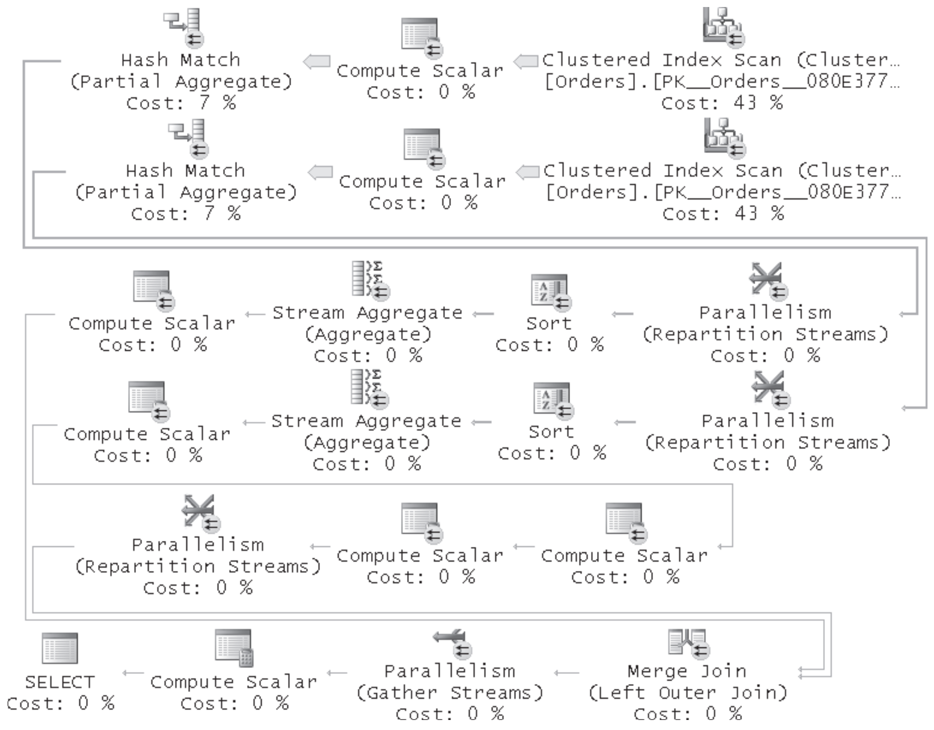
The first exercise involves multiple references to the same intermediate result set of a query. The task is to query the Orders table, and return for each order year the number of orders placed that year, and the difference from the number of orders placed in the previous year. Here's the desired output when you run your solution against InsideTSQL2008:

```
orderyear   numorders   diff
----------- ----------- -----------
2006        152         NULL
2007        408         256
2008        270         -138
```

You could use a table expression representing yearly counts of orders, and join two instances of the table expression to match to each current year the previous year, so that you can calculate the difference. Here's an example for implementing such an approach using a CTE:

```
SET STATISTICS IO ON;

WITH YearlyCounts AS
(
  SELECT YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
  FROM Sales.Orders
  GROUP BY YEAR(orderdate)
)
SELECT C.orderyear, C.numorders, C.numorders - P.numorders AS diff
FROM YearlyCounts AS C
  LEFT OUTER JOIN YearlyCounts AS P
    ON C.orderyear = P.orderyear + 1;
```

Remember that a table expression is nothing but a reflection of the underlying tables. When you query two occurrences of the *YearlyCounts* CTE, both get expanded behind the scenes. All the work of scanning the data and aggregating it happens twice. You can see this clearly in the query's execution plan shown in Figure 7-4.



Hash Match (Partial Aggregate) Cost: 7 %
Compute Scalar Cost: 0 %
Clustered Index Scan (Cluster… [Orders].[PK__Orders__080E377… Cost: 43 %
Hash Match (Partial Aggregate) Cost: 7 %
Compute Scalar Cost: 0 %
Clustered Index Scan (Cluster… [Orders].[PK__Orders__080E377… Cost: 43 %
Compute Scalar Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %
Sort Cost: 0 %
Parallelism (Repartition Streams) Cost: 0 %
Compute Scalar Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %
Sort Cost: 0 %
Parallelism (Repartition Streams) Cost: 0 %
Parallelism (Repartition Streams) Cost: 0 %
Compute Scalar Cost: 0 %
Compute Scalar Cost: 0 %
SELECT Cost: 0 %
Compute Scalar Cost: 0 %
Parallelism (Gather Streams) Cost: 0 %
Merge Join (Left Outer Join) Cost: 0 %

FIGURE 7-4 Execution plan for a solution to the "comparing periods" exercise (using table expressions)

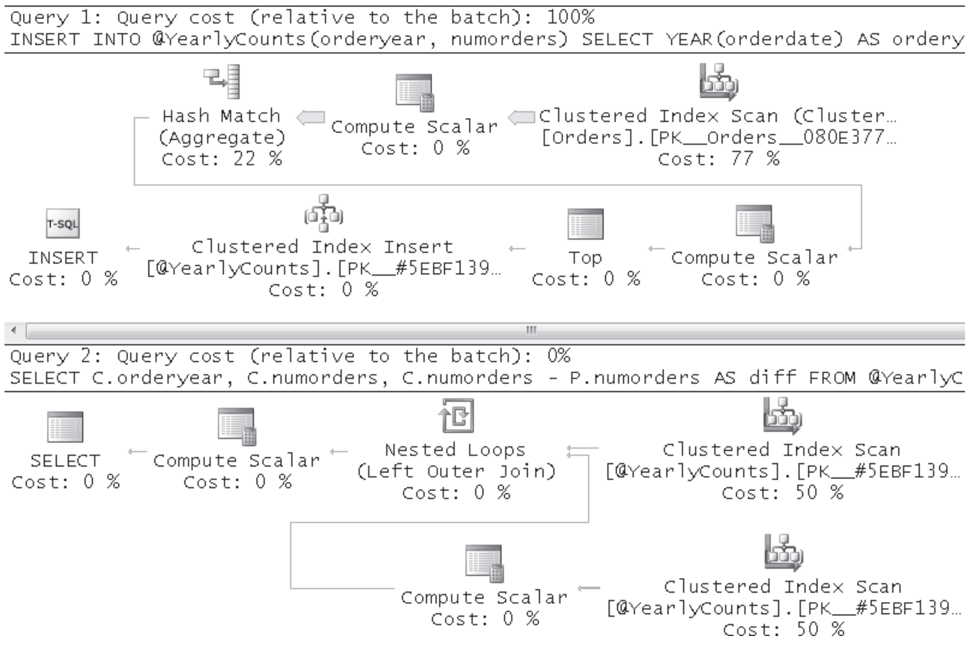
Scanning the base data from the clustered index involves 28,807 reads. Because the data was scanned twice, STATISTICS IO reports 57,614 reads. As you can realize, scanning and aggregating the base data twice is unnecessary. This is a scenario where you should consider using a temporary table or a table variable. When choosing between the two, remember that one of the things to consider is the size of the intermediate result set that will be stored in the temporary object. Because the intermediate result set here will have only one row per year, obviously it's going to be very tiny, and it will probably require only one or two pages. In this case, it makes sense to use a table variable and benefit from the fact that it will not cause plan optimality related recompiles.

Here's the solution using a table variable:

```
DECLARE @YearlyCounts AS TABLE
(
  orderyear INT PRIMARY KEY,
  numorders INT
);

INSERT INTO @YearlyCounts(orderyear, numorders)
  SELECT YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
  FROM Sales.Orders
  GROUP BY YEAR(orderdate);
```

```
SELECT C.orderyear, C.numorders, C.numorders - P.numorders AS diff
FROM @YearlyCounts AS C
  LEFT OUTER JOIN @YearlyCounts AS P
    ON C.orderyear = P.orderyear + 1;
```

The work that includes scanning the base data and aggregating it happens only once and the tiny result is stored in a table variable. Then the last query joins two instances of the tiny table variable to produce the desired output. The execution plan for this solution is shown in Figure 7-5.



```
Query 1: Query cost (relative to the batch): 100%
INSERT INTO @YearlyCounts(orderyear, numorders) SELECT YEAR(orderdate) AS ordery
```

```
Query 2: Query cost (relative to the batch): 0%
SELECT C.orderyear, C.numorders, C.numorders - P.numorders AS diff FROM @YearlyC
```

FIGURE 7-5 Execution plan for a solution to the "comparing periods" exercise (using table variables)

Because the base data from the clustered index on the Orders table was scanned only once, STATISTICS IO reports only about half the number of reads (28,621) compared to the previous solution. It also reports a very small number of reads (11) from the table variable.

Recent Orders

The task in the second exercise is to query the Orders table, and return for each customer the orders with the most recent order date for the customer. Here's the desired output when you run your solution against InsideTSQL2008, shown in abbreviated form:

```
orderid     orderdate               custid      empid
----------- ----------------------- ----------- -----------
11044       2008-04-23 00:00:00.000 91          4
11005       2008-04-07 00:00:00.000 90          2
11066       2008-05-01 00:00:00.000 89          7
10935       2008-03-09 00:00:00.000 88          4
```

```
11025       2008-04-15 00:00:00.000   87          6
11046       2008-04-23 00:00:00.000   86          8
10739       2007-11-12 00:00:00.000   85          3
10850       2008-01-23 00:00:00.000   84          1
10994       2008-04-02 00:00:00.000   83          2
10822       2008-01-08 00:00:00.000   82          6
...
```

There are many ways to solve this problem, some of which I'll present here. But most solutions benefit from the following index on *custid*, *orderdate* as the keys and *empid*, *orderid* as included columns:

```
CREATE INDEX idx_cid_od_i_eid_oid ON Sales.Orders(custid, orderdate)
  INCLUDE(empid, orderid);
```

The first solution I'll present is one where I use a CTE to calculate the maximum order date per customer, and then in the outer query join the Orders table with the CTE to return the orders with the maximum order date for each customer, like so:

```
WITH CustMax AS
(
  SELECT custid, MAX(orderdate) AS mx
  FROM Sales.Orders
  GROUP BY custid
)
SELECT O.orderid, O.orderdate, O.custid, O.empid
FROM Sales.Orders AS O
  JOIN CustMax AS M
    ON O.custid = M.custid
   AND O.orderdate = M.mx;
```

Here the fact that a table expression is not materialized—rather its definition gets expanded—is an advantage. You might expect SQL Server to scan the data twice—once to process the inner reference to the Orders table in the CTE query, and another for the outer reference to the Orders table. But the optimizer figured out a way to handle this query by scanning the data only once, which is truly admirable. Figure 7-6 shows the execution plan the optimizer produced for this query.

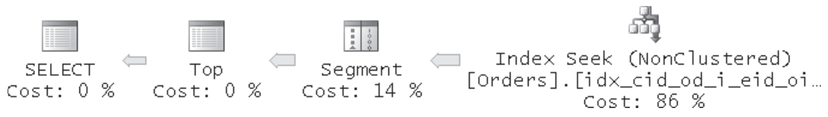


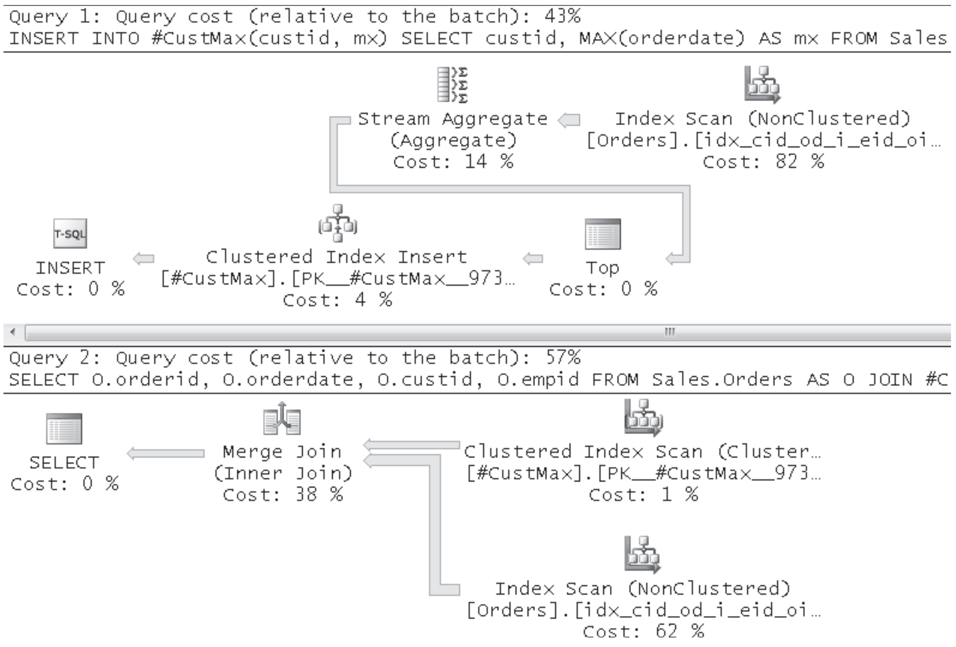
FIGURE 7-6 Execution plan for a solution to the "recent orders" exercise (using a CTE and join)

The Index Seek operator against the index you just created seeks the last entry in the leaf of the index, and then starts scanning the leaf level backward. The Segment operator segments the rows by customer, and the Top operator filters only the rows with the maximum order date per customer. This is a very efficient plan that requires scanning the index you created earlier only once, in order. STATISTICS IO reports 3,231 reads, which is close to the number of pages in the leaf of the index.

You realize that if you implement a similar solution, except using a temporary table instead of the table expression, the data will have to be scanned more than once—one time to produce the aggregated information you store in the temporary table, and another time to process the outer reference to Orders representing the base data that you join with the temporary table. Here's an implementation of this approach:

```
CREATE TABLE #CustMax
(
  custid INT      NOT NULL PRIMARY KEY,
  mx     DATETIME NOT NULL
);

INSERT INTO #CustMax(custid, mx)
  SELECT custid, MAX(orderdate) AS mx
  FROM Sales.Orders
  GROUP BY custid;

SELECT O.orderid, O.orderdate, O.custid, O.empid
FROM Sales.Orders AS O
  JOIN #CustMax AS M
    ON O.custid = M.custid
   AND O.orderdate = M.mx;

DROP TABLE #CustMax;
```

The execution plan for this solution is shown in Figure 7-7.



```
Query 1: Query cost (relative to the batch): 43%
INSERT INTO #CustMax(custid, mx) SELECT custid, MAX(orderdate) AS mx FROM Sales
```

Stream Aggregate
(Aggregate)
Cost: 14 %
Index Scan (NonClustered)
[Orders].[idx_cid_od_i_eid_oi…
Cost: 82 %
T-SQL
INSERT
Cost: 0 %
Clustered Index Insert
[#CustMax].[PK__#CustMax__973…
Cost: 4 %
Top
Cost: 0 %

```
Query 2: Query cost (relative to the batch): 57%
SELECT O.orderid, O.orderdate, O.custid, O.empid FROM Sales.Orders AS O JOIN #C
```

SELECT
Cost: 0 %
Merge Join
(Inner Join)
Cost: 38 %
Clustered Index Scan (Cluster…
[#CustMax].[PK__#CustMax__973…
Cost: 1 %
Index Scan (NonClustered)
[Orders].[idx_cid_od_i_eid_oi…
Cost: 62 %

FIGURE 7-7 Execution plan for a solution to the "recent orders" exercise (using temporary tables)

The first plan is for the population of the temporary table, and here you can see the first scan of the index you created earlier, plus aggregation of the data, and storing the result in the temp table's clustered index. The second plan is for the join query, showing that the base data from the index on Orders is scanned again, as well as the data from the temporary table, and the two inputs are joined using a merge join algorithm. STATISTICS IO reports twice 3,231 logical reads against Orders for the first plan, plus 3,231 logical reads against Orders and 28 logical reads against the temporary table for the second plan.

Clearly, in this case, the approach using the table expression was more efficient. By the way, this problem has other solutions using table expressions. For example, the following solution uses the CROSS APPLY operator and a derived table:

```
SELECT A.*
FROM Sales.Customers AS C
  CROSS APPLY (SELECT TOP (1) WITH TIES orderid, orderdate, custid, empid
               FROM Sales.Orders AS O
               WHERE O.custid = C.custid
               ORDER BY orderdate DESC) AS A;
```

Figure 7-8 shows the execution plan for this query.

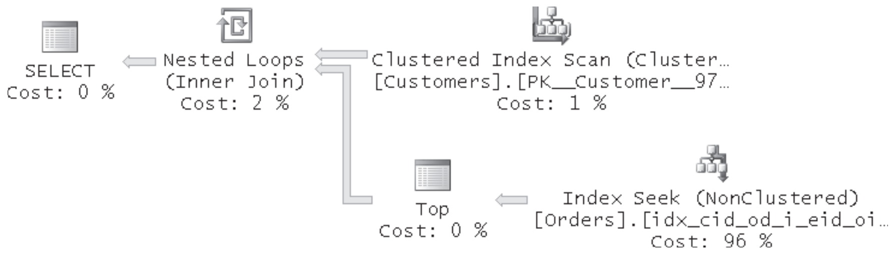

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 2 %
Clustered Index Scan (Cluster…
[Customers].[PK__Customer__97…
Cost: 1 %
Top
Cost: 0 %
Index Seek (NonClustered)
[Orders].[idx_cid_od_i_eid_oi…
Cost: 96 %

FIGURE 7-8 Execution plan for a solution to the "recent orders" exercise (using a derived table and APPLY)

As you can see, the plan scans the clustered index on the Customers table, and for each customer, uses a seek operation against the nonclustered index on Orders to pull the orders that were placed by the current customer in its maximum order date. With a low density of customers—as in our case—this plan is less efficient than the one shown in Figure 7-7 for the previous solution based on a CTE. STATISTICS IO reports 31,931 reads from Orders, and those are random reads unlike the sequential ones you got from the plan in Figure 7-7. The solution based on the APPLY operator excels particularly when the density of customers is very high, and the number of seek operations is therefore accordingly small.

Finally, another solution based on table expressions that you might want to consider is one that assigns ranks to orders partitioned by customer—ordered by order date descending—and then filter only the rows with a rank value equal to 1. For a partitioned ranking calculation, the optimizer will only use an index and avoid sorting if the key columns have the same sorting direction in the index as they do in the ranking calculation's OVER clause. Create a nonclustered index with *orderdate* descending like so:

```
CREATE INDEX idx_cid_od_i_eid_oidD ON Sales.Orders(custid, orderdate DESC)
  INCLUDE(empid, orderid);
```

Then you will get an efficient plan from the following solution:

```
WITH OrderRanks AS
(
  SELECT orderid, orderdate, custid, empid,
    RANK() OVER(PARTITION BY custid ORDER BY orderdate DESC) AS rnk
  FROM Sales.Orders
)
SELECT *
FROM OrderRanks
WHERE rnk = 1;
```

The plan is shown in Figure 7-9.

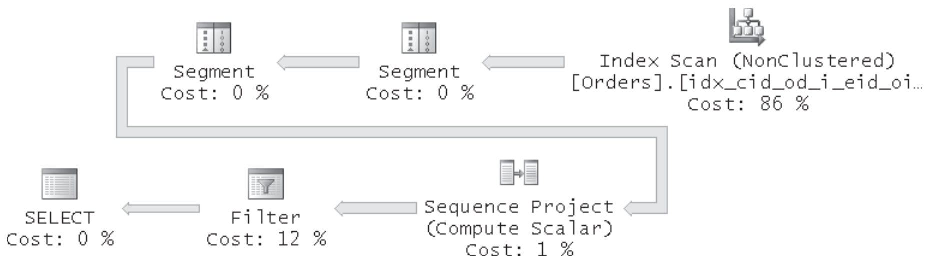


FIGURE 7-9 Execution plan for a solution to the "recent orders" exercise (using ranks and a CTE)

The efficiency of the plan is quite similar to the one shown earlier in Figure 7-7. Here as well the nonclustered index is scanned once in order. STATISTICS IO reports 3,231 logical reads as expected. This plan, like the one shown in Figure 7-7, excels when the density of customers is low.

When you're done, run the following code for cleanup:

```
DROP INDEX Sales.Orders.idx_cid_od_i_eid_oid;
DROP INDEX Sales.Orders.idx_cid_od_i_eid_oidD;
```

Relational Division

For the last summary exercise, you're given the following task: you need to determine which customers have orders handled by the same set of employees. The result set should contain one row for each customer, with two columns: the customer ID and a value that identifies the group of employees that handled orders for the customer. The latter is expressed as the minimum customer ID out of all customers that share the same group of employees. That is, if customers A, B, and D were handled by one group of employees (for example, 3, 7, 9), and customers C and

E by a different group (for example, 3 and 7), the result set would contain {(A, A), (B, A), (D, A), (C, C), (E, C)}. It will be convenient to use NULL instead of the minimum customer ID to identify the group of no employees for customers without orders. Following is the desired result against the InsideTSQL2008 database, shown here in abbreviated form:

```
custid      grp
----------- -----------
22          NULL
57          NULL
1           1
2           2
78          2
3           3
81          3
4           4
5           5
34          5
...
```

You can observe, for example, that customers 2 and 78 were handled by the same group of employees, because for both customers, *grp* is 2. Remember that you should use the sample tables in the InsideTSQL2008 database only to check the accuracy of your result. For performance estimations, use the tables you created earlier in tempdb by running the code in Listing 7-1. Like before, also with this problem the performance measures I will mention were measured against the tables in tempdb.

The first solution doesn't make any use of temporary objects; rather, it implements a classic relational division approach applying reverse logic with subqueries:

```
SELECT custid,
    CASE WHEN EXISTS(SELECT * FROM Sales.Orders AS O
                     WHERE O.custid = C1.custid)
      THEN COALESCE(
        (SELECT MIN(C2.custid)
           FROM Sales.Customers AS C2
           WHERE C2.custid < C1.custid
             AND NOT EXISTS
               (SELECT * FROM Sales.Orders AS O1
                WHERE O1.custid = C1.custid
                  AND NOT EXISTS
                    (SELECT * FROM Sales.Orders AS O2
                     WHERE O2.custid = C2.custid
                       AND O2.empid = O1.empid))
             AND NOT EXISTS
               (SELECT * FROM Sales.Orders AS O2
                WHERE O2.custid = C2.custid
                  AND NOT EXISTS
                    (SELECT * FROM Sales.Orders AS O1
                     WHERE O1.custid = C1.custid
                       AND O1.empid = O2.empid))),
        custid) END AS grp
FROM Sales.Customers AS C1
ORDER BY grp, custid;
```

The query invokes a CASE expression for every customer from the Customers table (C1). The CASE expression invokes the COALESCE function for customers who placed orders, and returns NULL for customers who placed no orders. If the customer placed orders, COALESCE will substitute a NULL returned by the input expression with the current *custid*. The input expression will return the result of the following:

- Return the minimum *custid* from a second instance of Customers (C2)
- Where *C2.custid* (*cust2*) is smaller than *C1.custid* (*cust1*)
- And you cannot find an employee in *cust1*'s orders that does not appear in *cust2*'s orders
- And you cannot find an employee in *cust2*'s orders that does not appear in *cust1*'s orders

Logically, you could do without filtering *cust2 < cust1*, but this expression is used to avoid wasting resources. Anyway, you need to return the minimum *custid* out of the ones with the same employee list. If customer A has the same employee group as customer B, both will end up with a *grp* value of A. For customer B, there's a point in comparing it to customer A (smaller ID), but for customer A there's no point in comparing it to customer B (higher ID). Naturally, the minimum *custid* with a given employee group will not have the same employee group as any customers with smaller IDs. In such a case, the expression will return NULL, and the outer COALESCE will substitute the NULL with the current *custid*. As for the rest, it's a classical phrasing of relational division with reverse logic.

This solution is expensive because of the excessive scan count, which has to do with the large number of invocations of the correlated subqueries. To give you a sense, this solution ran over an hour before I gave up waiting for it to finish and stopped it. Most standard set-based solutions you can come up with for this problem that don't use temporary objects will typically be expensive.

If you devise a solution in which you generate an interim set that can benefit from an index, you might want to consider using temporary tables. For example, you can materialize the distinct list of *custid*, *empid* values; index the temporary table; and continue from there. The materialized data would substantially reduce the number of rows in the set you'll query. Still, you won't be dealing with a tiny set, and most probably your solution will access the table multiple times. You want efficient plans to be generated based on distribution statistics and accurate cardinality information. All this should lead you to use a local temporary table and not a table variable.

Here's a solution that first creates the suggested local temporary table, indexes it, and then queries it:

```
SELECT DISTINCT custid, empid
INTO #CustsEmps
FROM Sales.Orders;
```

```
CREATE UNIQUE CLUSTERED INDEX idx_cid_eid
  ON #CustsEmps(custid, empid);
GO

WITH Agg AS
(
  SELECT custid,
    MIN(empid) AS MN,
    MAX(empid) AS MX,
    COUNT(*)   AS CN,
    SUM(empid) AS SM,
    CHECKSUM_AGG(empid) AS CS
  FROM #CustsEmps
  GROUP BY custid
),
AggJoin AS
(
  SELECT A1.custid AS cust1, A2.custid AS cust2, A1.CN
  FROM Agg AS A1
    JOIN Agg AS A2
      ON  A2.custid <= A1.custid
      AND A2.MN = A1.MN
      AND A2.MX = A1.MX
      AND A2.CN = A1.CN
      AND A2.SM = A1.SM
      AND A2.CS = A1.CS
),
CustGrp AS
(
  SELECT cust1, MIN(cust2) AS grp
  FROM AggJoin AS AJ
  WHERE CN = (SELECT COUNT(*)
              FROM #CustsEmps AS C1
                JOIN #CustsEmps AS C2
                  ON C1.custid = AJ.cust1
                  AND C2.custid = AJ.cust2
                  AND C2.empid = C1.empid)
  GROUP BY cust1
)
SELECT custid, grp
FROM Sales.Customers AS C
  LEFT OUTER JOIN CustGrp AS G
    ON C.custid = G.cust1
ORDER BY grp, custid;
GO

DROP TABLE #CustsEmps;
```

I also used CTEs here to build the solution in a modular approach. The first CTE (*Agg*) groups the data from the temporary table by *custid*, and returns several aggregates based on *empid* for each customer (MIN, MAX, COUNT, SUM, CHECKSUM_AGG).

The second CTE (*AggJoin*) joins two instances of *Agg* (*A1* and *A2*)—matching each customer in *A1* to all customers in *A2* with a lower *custid* that have the same values for all the aggregates.

The purpose of comparing aggregates is to identify pairs of customers that *potentially* share the same group of employees. The reasoning behind the use of less than or equal to (<=) in the filter is similar to the one in the previous solution. That is, comparing groups of employees between customers when *A2.custid (cust2)* is greater than *A1.custid (cust1)* is superfluous.

The third CTE, (*CustGrp*), filters from *AggJoin* only pairs of customers that actually share the same group of employees, by verifying that the count of matching employees in both groups is identical to the total count of employees in each group by itself. The query aggregates the filtered rows by *cust1*, returning the minimum *cust2* for each *cust1*. At this point, *CustGrp* contains the correct *grp* value for each customer.

Finally, the outer query performs a left outer join that adds customers without orders.

This solution runs for eight seconds. Note that you could use a CTE with the set of distinct *custid, empid* combinations instead of the temporary table #CustEmps. This way, you could avoid using temporary tables altogether. I tested such a solution and it ran for about 12 seconds—50 percent more than the solution that utilizes a temporary table. The advantage in the temporary table approach was that you could index it.

Considering the fastest solution we had so far—the one utilizing a temporary table—is this really the best you can get? Apparently not. You can use the FOR XML PATH option to concatenate all distinct *empid* values per customer. You can then group the data by the concatenated string, and return for each customer the minimum *custid* within the group using the OVER clause. The fast and nifty concatenation technique was originally devised by Michael Rys and Eugene Kogan. The PATH mode provides an easier way to mix elements and attributes than the EXPLICIT directive. Here's the complete solution:

```
WITH CustGroups AS
(
  SELECT custid,
    (SELECT CAST(empid AS VARCHAR(10)) + ';' AS [text()]
     FROM (SELECT DISTINCT empid
           FROM dbo.Orders AS O
           WHERE O.custid = C.custid) AS D
     ORDER BY empid
     FOR XML PATH('')) AS CustEmps
  FROM dbo.Customers AS C
)
SELECT custid,
  CASE WHEN CustEmps IS NULL THEN NULL
    ELSE MIN(custid) OVER(PARTITION BY CustEmps) END AS grp
FROM CustGroups
ORDER BY grp, custid;
```

The solution is short and slick, doesn't use temporary tables at all, and runs for six seconds!

Conclusion

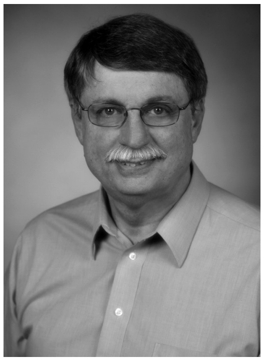
I hope this chapter helped you realize the significant differences between the various types of temporary objects supported by SQL Server. I had to dispel a few widespread myths, especially with regard to table variables. Remember that it's typically advisable to use table variables for small tables and to compare table variables against temporary tables for the most critical queries. You realize that there's a time and place for each type and that no one type is always preferable to the others. I gave you a summary table with the aspects and functionality of each type, which should help you make the right choices based on your needs. Also, remember to pay special attention to tempdb, which can become a bottleneck in your system, especially when working extensively with temporary tables.

Index

About the Authors

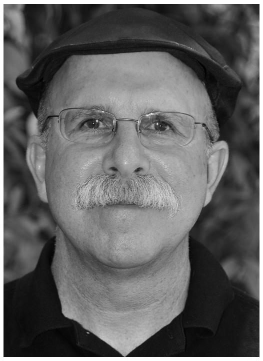
Itzik Ben-Gan is a mentor and cofounder of Solid Quality Mentors. A SQL Server Microsoft MVP (Most Valuable Professional) since 1999, Itzik has delivered numerous training events around the world focused on T-SQL querying, query tuning, and programming. Itzik is the author of several books about T-SQL. He has written many articles for *SQL Server Magazine* as well as articles and white papers for MSDN. Itzik's speaking engagements include Tech Ed, DevWeek, PASS, SQL Server Magazine Connections, various user groups around the world, and Solid Quality Mentors events.

Dejan Sarka focuses on development of database and business intelligence applications. Besides projects, he spends about half of the time on training and mentoring. He is a frequent speaker on some of the most important international conferences such as PASS, TechEd, and SqlDevCon. He is also indispensable on regional MS events, for example on the NT Conference, the biggest MS conference in Central and Eastern Europe. He is the founder of the Slovenian SQL Server and .NET Users Group. Dejan Sarka is the main author, coauthor, or guest author of seven books about databases and SQL Server. Dejan Sarka also developed two courses for Solid Quality Learning—Data Modeling Essentials and Data Mining with SQL Server 2008.



Roger Wolter is an architect on the Microsoft IT MDM project team. He has 30 years of experience in various aspects of the computer industry including jobs at Unisys, Infospan, Fourth Shift, and the last 10 years as a Program Manager at Microsoft. His projects at Microsoft include SQLXML, the Soap Toolkit, the SQL Server Service Broker, SQL Server Express, and Master Data Services. He is currently working on a project to master all of Microsoft's customer and partner data.

Greg Low is a consultant and trainer, best known for his SQL Down Under podcast *(www.sqldownunder.com)* and for his work as the director of global chapter operations for PASS. Greg is the country lead for Solid Quality Mentors in Australia (*www.solidq.com.au*), a SQL Server MVP, and a Microsoft Regional Director. He holds a PhD in computer science from QUT and has written a number of books on SQL Server and on building technical communities. For Microsoft, he has written SQL Server white papers and training materials and is one of a handful of trainers chosen to deliver the Microsoft Certified Masters program for SQL Server 2008.



Ed Katibah is a program manager on the Microsoft SQL Server Strategy, Infrastructure and Architecture team. Ed began his professional career over 34 years ago while working in a University of California, Berkeley, research group at the Space Sciences Laboratory. Ed has extensive experience in the spatial industry with jobs ranging from research, software development, consulting, application programming, and large-scale spatial database production systems. Since 1996, Ed has worked exclusively on spatially enabled database systems for Informix, IBM, and now Microsoft.

Isaac Kunen is a program manager on the SQL Server engine programmability team. Since joining Microsoft in 2005, he has worked on the type system, SQL CLR integration, and database extensibility. His most prominent project to date is the spatial data support in SQL Server 2008. Isaac is currently focusing on reducing the complexity of database application development, deployment, and management.